Fachhochschul-Masterstudiengang
SOFTWARE ENGINEERING
4232 Hagenberg, Austria

BoB: Best of Both in Compiler Construction

Combination of Bottom-up Syntax Analysis and Top-down Semantic Analysis

Masterarbeit

zur Erlangung des akademischen Grades
Master of Science in Engineering

Eingereicht von

Ing. Wolfgang Dichler BSc

Begutachter: FH-Prof. DI Dr. Heinz Dobler

Juni 2013

software engineering
IKM Fakultät Hagenberg

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

University of Applied Sciences Hagenberg, June 18, 2013

Ing. Wolfgang Dichler, BSc

Contents

Kurzfassung

Diese Masterarbeit beschäftigt sich mit dem Problem, dass aktuelle Compiler entweder Top-down- oder Bottom-up-Syntaxanalyse verwenden. Beide Konzepte haben Nachteile. Die Bottom-up-Analyse, für die mächtigen LR(k)-Grammatiken, unterstützt nur S- oder LR-attributierte Grammatiken. Im Gegensatz dazu unterstützt die Top-down-Analyse, für die LL(k)-Grammatiken, die mächtigeren L-attributierten Grammatiken.

Das Ziel dieser Masterarbeit ist, die Vorteile der Bottom-up-Syntaxanalyse mit den Vorteilen der Top-down-Semantikauswertung zu kombinieren. Hierfür wird ein Präprozessor namens *BoB* (*Best of Both*) für *flex* und *bison* entwickelt.

BoB verarbeitet Compiler-Beschreibungen – geschrieben in *Cocol4BoB* – und erlaubt die Verarbeitung von L-attributierten LALR(1)-Grammatiken. Die durch die neue *BoB-flex-bison*-Werkzeugkette erzeugten Compiler verwenden Bottom-up-Syntaxanalyse und Top-down-Semantikauswertung.

Die Stärke von *BoB* ist die einfache und leistungsfähige Compiler-Beschreibungssprache *Cocol4BoB*. Entwickler/innen müssen sich nicht um LL(1)-Konflikte kümmern und können in den semantischen Aktionen Eingangs-, Ausgangs- und Übergangsattribute verwenden. Deshalb ist der Einstieg für Anfänger/innen bei *BoB* besonders einfach möglich. Mit diesen Vorteilen ist leider ein Nachteil verbunden: Compiler die durch *BoB* erzeugt werden haben eine längere Laufzeit.

In dieser Arbeit wird gezeigt, dass die Konzepte von *BoB* Vorteile in Bezug auf die Benutzerfreundlichkeit und Mächtigkeit der Eingabesprache haben. Trotzdem sollten die offenen Fragen bezüglich der Performance nicht vergessen und in zukünftigen Arbeiten behandelt werden.

Abstract

This master thesis deals with the problem, that current compilers use either top-down or bottom-up syntax analysis. Both concepts have their disadvantages. Bottom-up analysis for the powerful LR(k) grammars only supports S- or LR-attributed grammars. On the other hand, top-down analysis for the less powerful LL(k) grammars supports the powerful L-attributed grammars.

The goal of this master thesis is to combine the advantages of syntax analysis and semantic evaluation. Therefore, a preprocessor called *BoB* (Best of Both) for *flex* and *bison* is implemented.

BoB process a compiler description written in *Cocol4BoB* and allows L-attributed LALR(1) grammars. Compilers generated by the new *BoB-flex-bison* toolchain use bottom-up syntax analysis and top-down semantic evaluation.

The strength of *BoB* is the simple and powerful compiler description language *Cocol4BoB*. Developers do not have to handle LL(1) conflicts. Moreover, they can use inherited and synthesized attributes in semantic actions. Therefore, the usage for beginners is less difficult than in other compiler generators. However, these advantages came with a drawback: compilers generated by *BoB* have a longer compile-time.

This thesis shows, that the concepts of *BoB* have advantages in terms of usability and capability of the input language. Nevertheless, the open performance issues should be addressed in further projects.

Chapter 1

Introduction

Compilers typically are used to translate programs from higher-level languages to a lower-level language. The executable machine code builds the bottom of the language layers. This code can be directly executed by the used hardware or virtual machine.

Software development has changed in the last decades. The introduced compilers allow the programmer to develop software in an abstract way. So the software gets independent from the underlying hardware. Newer compiler/language features added further abstraction layers, which help to reduce the error probabilities and increase the system maintainability. Therefore, compilers are the most important tools for software developers.

In the past, compilers were usually developed as part of the invention of new programming languages. Today, developers do not only use compilers, they develop their own compilers for custom, domain-specific languages (DSLs).

1.1 Background and Motivation

At the beginning, compilers were written by hand without any formal specification. Later compilers were created from an attributed grammar (ATG). For instance, they used the method of recursive descent for parsing. The developments in the last decades improved this error-prone transformation. For this purpose, tools have been developed which support the developers by generating parts of their compilers. This tools are called compiler compilers or compiler generators. These terms are misleading because they imply that a full compiler can be created. Usually, these tools create only parts, such as the scanner or the parser.

There are two main categories of compiler generators. The first category uses for syntax and semantic analysis top-down parsers. The second one uses bottom-up parsers. The top-down parsers can process the less powerful LL(k)

grammars [SSS90]. Bottom-up parsers support the class of LR(k) grammars [SSS90].

The drawback that top-down parsers can only process LL(k) grammars is caused by the less information available during the processing of the input stream. The parser has to decide which rule to derive. In contrast, the bottom-up parsers push the terminal symbols onto a stack and they reduce only when a rule matches. On the other side, bottom-up parsers have the disadvantage that they can not process inherited and transitional attributes. Since the processing takes place from bottom upwards, no information of the formal parameter list will be known.

Both approaches have their drawbacks. Therefore, compiler developers have always to accept a compromise. In 1986, Hanspeter Mössenböck described in his dissertation *Compilererzeugende Systeme für Mikrocomputer* [Mös87] a new compiler generator called *Smart* which overcomes these limitations (see section 2.3.3). This compiler generator is written in and for *Modula-2* [Wir85]. Nowadays, this programming language has no large spread [TIO13], thus the compiler generator has not been used on a wide scale.

1.2 Problem Definition

Current compiler generators have significant disadvantages in the ease of use. Either the compiler generators do not support LR(k) grammars, or they do not support inherited and transitional attributes in semantic actions. Therefore, the users have to choose between bottom-up and top-down parsers and have to deal with their drawbacks.

This work deals with a solution to this problem. The focus does not lie in the development of a new compiler generator. It lies in the extension or adaptation of existing compiler generators to solve the problem. A new preprocessor *BoB* (**B**est **o**f **B**oth) for *flex* [PEM07] and *bison* [DS11] should be implemented. The resulting tool-chain *BoB-flex-bison* should be able to generate compilers written in C++ from attributed LR(k) grammars which support inherited and transitional attributes.

1.3 Aims and Objectives

The overall aim of this work is to combine the best ways for parsing and semantic analysis. The powerful bottom-up parsing with the powerful top-down semantic evaluation.

The first part of the work is to analyze the state of the art of compiler construction and acquiring knowledge about the tools (*flex* and *bison*).

Moreover, existing approaches of bottom-up and top-down parsing should be analyzed and checked regarding to their suitability. Based on this preliminary work, a preprocessor should be implemented. This preprocessor should use an ATG syntax inspired by *Cocol-2* syntax (see [DP90]), where for the syntax LALR(1) grammars are allowed while for the semantic actions only the restriction to L-attribution exists.

The preprocessor *BoB* should translate an ATG into valid input files for *flex* and *bison*. These compiler generators should then be used to generate the final compiler. The resulting compiler should implement bottom-up parsing and top-down semantic evaluation. For realizing the semantic evaluator the ideas of *Smart* should taken up again.

To develop *BoB* prior knowledge in the area of compiler construction is necessary, this can, e.g., be found in [Aho+06]. In addition, the basics of the programming language C++ are required, [Pra11] offers a good insight.

1.3.1 Functional Requirements

During the design analysis, the following functional requirements are identified for the preprocessor *BoB*:

- A new compiler description language called *Cocol4Bob*. *Cocol4BoB* should be similar to *Cocol-2*, but must fulfill the requirements of the target language C++ and the necessary changes by *flex* and *bison*.
- *BoB* must transform the lexical and syntactical parts of the input file to input files for *flex* and *bison*. From these generated files a bottom-up parser must be generated.
- *BoB* must generate a semantic evaluator on the basis of the semantic parts of the input. The semantic evaluator must process the input in a top-down manner.
- *BoB* should provide an error recovery and must provide a error notification mechanism. The error notification mechanism has to inform the user about the lexical, syntactic and semantic errors during the processing. The user should get a detailed report about the errors. This includes the error position in the source document and a error description.
- *BoB* should have a command-line mode and an interactive mode. The command-line mode is important to automate the tool-chain via scripts. The interactive mode can be used by the users to get more feedback about the transformation process.
- The compilers generated by *BoB* must be able to be build with common available compilers like *GNU gcc* or the *Microsoft C/C++* compiler.

1.3.2 Non-Functional Requirements

Moreover, there are some additional non-functional requirements that should be fulfilled by *BoB*:

- *BoB* itself and compilers generated by *BoB* should have a modular architecture for simple exchange of components.
- The constant parts of the generated files should be provided by templates. These templates offer the possibility to take influence on the generated files.
- The run-time and memory consumption should be as low as possible.
- The generated *flex* and *bison* files should have readable structures. This opens the possibility to take only portions of the generated files.
- The resulting compiler should use the concepts introduced by the C++11 standard.

1.4 Structure of Thesis

Chapter 2 (State of the Art) treats the basic technologies, which are needed in the course of this work. First, the different types of attributed grammars are explained. Next, a brief introduction to the components of a compiler front-end is given. After that, the compiler generators which are important for this work are presented. Finally, the relevant compiler description languages are described.

Chapter 3 (Design) provides an overview of the selected solution approach. This includes the specification of the used compiler description language *Cocol4BoB*, the architecture of *BoB* itself and the architecture of compilers generated by *BoB*.

Chapter 4 (Implementation) treats the implementation of the design described in Chapter 3.

Chapter 5 (Tests, Evaluation, and Discussion) covers three topics. First, the test procedure and results are explained. After that, *BoB* is evaluated and compared with similar compiler generators. Finally, a the chosen solution approach is discussed.

Chapter 6 (Summary, Outlook and Experience) summarizes the work and gives an outlook on further developments. In addition, this chapter gives an insight into the personal experiences of the author, that were collected during the elaboration of this thesis.

Appendix A contains a *Cocol4BoB* grammar of *BoB* which can be used for bootstrapping.

Chapter 2

State of the Art

This chapter covers the basic principles and concepts that are needed in the course of this work. The first part deals with the fundamentals of formal languages. Especially, deterministic grammars are discussed in detail. The next part explains the general structure of a compiler front-end. Afterwards, the compiler generators, which are important for this work, are presented. The chapter ends with an overview of the compiler description languages that serve as foundation for this work.

2.1 Attributed Grammars (ATGs)

In [Dob10] ATGs are described as algorithms, which are formulated in an abstract or formal way. They are used to describe a syntax-driven translation processes in the field of compiler construction, e.g., as input for a compiler generator. The base of an ATG is a **context-free grammar** which is extended by

- **attributes** which represent semantic values of (non-)terminal symbols and
- **semantic actions** which are instructions to calculate those attributes.

The attributes of non-terminal symbols are classified into inherited attributes, synthesized/derived attributes and transitional attributes. Attributes of terminal symbols are always synthesized/derived attributes and are called lexical attributes. These attributes are calculated by the lexical analyzer (scanner). All other attributes are calculated during semantic evaluation.

The semantic actions for calculating attribute values are weaved into the right side of the grammar rules. These semantic actions are usually notated in a conventional programming language, e.g., the target language. The source code example below shows a grammar rule *Expr* which calculates the result of the attribute e within the semantic actions. In the semantic actions the programming language C++ is used.

Cocol4BoB

```
1 Expr<|int &e|> =                  LOCAL<|int t = 0;|>
2   Term<|e|>
3   { '+' Term<|t|>                 SEM<|e = e + t;|>
4   | '-' Term<|t|>                 SEM<|e = e - t;|>
5   }.
```

2.1.1 LL(k) Grammars

LL(k) grammars are processed from **L**eft to right with a **L**eft canonical derivation and with **k** symbols lookahead. LL(k) grammars are suitable for top-down recognition. It is necessary to distinguish between strong and weak LL(k). [Dob11] gives the following definition for the strong LL(k) condition:

> A grammar is "strong" LL(k) if each of its rules $A \to \alpha_1 | \ldots | \alpha_n$ and for all $1 \leq i < j \leq n$ applies:
>
> $$First_k(\alpha_i Follow_k(A)) \cap First_k(\alpha_j Follow_k(A)) = \{\}$$

This definition is too strict, since the set $Follow_k(A)$ is to general. A sufficient definition which use local follow-sets is defined by [Dob11] as below:

> A grammar is ("weak") LL(k) if each of its rules $A \to \alpha_1 | \ldots | \alpha_n$, each sential form which contains A $(S \Rightarrow^* \omega_1 A \omega_2)$ and for all $1 \leq i < j \leq n$ applies:
>
> $$First_k(\alpha_i \omega_2) \cap First_k(\alpha_j \omega_2) = \{\}$$

Top-down parsers usually use a LL(1) grammars. This has the advantage that the parsers can choose the production of an non-terminal symbol just by using the current input symbol. Moreover LL(1) grammars are sufficient for a wide range of applications. However, left recursive or ambiguous grammars can not be recognized.

2.1.2 LR(k) Grammars

LR(k) grammars are a proper superset of the LL(k) grammars. LR(k) grammars are processed from **L**eft to right with a reverse **R**ight derivation (inverse derivation = reduction) and with **k** symbols lookahead. [Dob11] defines the LR(k) condition as follow:

A grammar is LR(k) if from the three conditions

1. $S \overset{R}{\Rightarrow}^{*} \omega_1\ A\ \omega_2 \overset{R}{\Rightarrow} \omega_1\ \alpha\ \omega_2$
2. $S \overset{R}{\Rightarrow}^{*} \tau_1\ B\ \tau_2 \overset{R}{\Rightarrow} \omega_1\ \alpha\ \omega_3$
3. $First_k(\omega_2) = First_k(\omega_3)$

follows, that

1. $A = B$,
2. $\omega_1 = \tau_1$ and
3. $\tau_2 = \omega_3$

applies.

LR(k) grammars are processed by bottom-up parsers, also called shift/reduce parsers. The shift step reads the next input symbol and puts it on the stack. The reduce step applies a grammar rule on the current stack contents.

2.1.3 S- and LR-attributed Grammars

S-attributed grammars allow only synthesized attributes. This means that all attributes only depend on attributes of child nodes of the syntax tree. This limitation permits the flow of information from the bottom of the syntax tree upwards only. Therefore, this class is predestined for bottom-up parsers, since they have the same control flow through the syntax tree. S-attributed grammars are used in compiler-generators like *yacc* or *bison*. They do not use a clean S-attributed grammar, because inherited attributes could be imitated by using global variables. This solution goes in the direction of the LR-attributed grammars.

LR-attributed grammars allow the evaluation of inherited attributes during LR-parsing. This approach is implemented by *A preprocessor for YACC* [Kat83] and by the compiler generator *Zyacc* [Zer97]. [Zer97] describes the LR-attributed grammars as a superset of S-attributed grammars and a subset of L-attributed grammars. There are many solutions for handling LR-attributed grammars, depending on the implementation, there are larger or smaller limitations. A comparison of the different LR-attributed grammar types is given in [AMT90].

2.1.4 L-attributed Grammars

L-attributed grammars allow attribute evaluation in one left to right traversal without an explicit syntax tree. Within L-attributed grammars, inherited

Table 2.1: Example: Rule that is not L-attributed [Aho+06]

production	semantic actions
$A_{\updownarrow s} \rightarrow B_{\uparrow b \downarrow i}\ C_{\uparrow c}$	A.s = B.b; B.i = f(C.c, A.s);

and synthesized attributes are allowed. The only restriction is that inherited attributes depend on attributes which are calculated to the left of the corresponding non-terminal symbol. Due to this characteristics, this class is ideal for top-down parsing. This class enables the control flow of the attribute data in both directions. [Dob10] gives the following definition of L-attributed grammars:

> An ATG is L-attributed if all of its rules $A \rightarrow X_1 \ldots\ X_n$ meet the requirement that the input arguments of X_i for all $1 \leq i \leq n$ depend only on the inherited attributes of A and the synthesized attributes of X_1 ... X_{i-1}.

The example in Table 2.1 illustrates the L-attribution. The first semantic action `A.s = B.b` is allowed in S- and L-attributed grammars. In this case `A.s` represents a synthesized attribute, which is built from child attributes only. The second semantic action `B.i = f(C.c,A.s)` defines an inherited attribute, therefore, it is not S-attributed. Moreover, the action is not L-attributed because the inherited attribute `B.i` is dependant from attribute `C.c`. Since `C` comes right from `B`, it is not permitted.

2.2 Compiler Front-End

Modern compilers typically consist of a front-end and a back-end. The front-end handles the details of the source language and the back-end handles the target language. For communication between these two components an intermediate representation (IR) is used. This IR ensures the clean separation of these two components. This is important, because it allows the exchange of the front- and back-ends. The developers have to develop only m front-ends and n back-ends to generate mxn compilers [Aho+06].

For this work, only the front-end is relevant. As shown in Figure 2.1 the front-end can be divided into three phases – the lexical analysis, the syntactical analysis and the semantic analysis. These phases are discussed in the next sections in more detail. These sections are based on [Aho+06] which can serve as a good starting point for more detailed research.

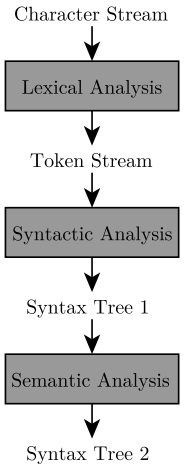


Figure 2.1: Phases of a front-end [Aho+06]

2.2.1 Lexical Analysis

The task of lexical analysis is to transform the input stream (the source code) into a stream of tokens. While reading the input stream, meaningful character sequences – called lexemes – are detected. A lexeme is defined by a pattern, e.g., by regular expressions or *Extended Backus-Naur Form* (EBNF). Each lexeme represents an instance of a token. These tokens consist of a name and an optional attribute value. The recognized tokens are delivered to the syntactic analysis for further processing.

The lexical analysis includes some additional tasks: One important task is to ignore the irrelevant parts of the source code. These are the separators of the lexemes (e.g., white space and line feed) and comments. Another task is the special treatment of comments, for example the creation of a source code documentation from the content within the comments.

Another point of the lexical analysis is the treatment of pragma directives. These are instructions, e.g., to include source code, carry out macro expansion and perform conditional compilation. Depending on the complexity of the supported directives, a preprocessor may be required.

One additional task is the mapping of error messages to the underlying source code. Therefore, the current source document position has to be tracked. From the tracked line and column numbers meaningful error messages can be delivered.

2.2.2 Syntactic Analysis

The syntactic analysis checks the syntactic correctness of the input. The syntactic analysis processes the token stream delivered from the lexical analysis and creates a tree-based IR. This represents the grammatical structure of the syntax tree. During processing of the tokens, it is checked whether the grammar of the source language allows the current token. Otherwise, syntax errors are saved and finally delivered to the users. The grammar of the source language is usually defined by a syntax description language like *Backus-Naur Form* (BNF) or EBNF.

The parse tree described above is usually not created explicitly, because the processing of the parser is woven with the other steps of the front-end. The syntax tree is only implicitly created by the processing order.

One distinguishes between top-down and bottom-up parsers: Top-down parsers build the parse tree starting at the root to the leaves. The creation order of the nodes corresponds to a pre-order traversal of the tree. Bottom-up parsers build the parse tree starting at the leaves up to the root. This corresponds to a post-order creation sequence. These two types of parsers are suitable for certain sub-types of grammars only. For the top-down analysis it is the LL-grammar and for bottom-up analysis it is the LR-grammar (see section 2.1).

For a reasonable syntax analysis, it is crucial to provide meaningful error messages. Moreover, an adequate error recovery mechanism is required to allow a user-friendly error handling.

2.2.3 Semantic Analysis

The semantic analysis takes care of the semantics of the inputs. During semantic analysis, static conditions are checked that go beyond the lexical and syntax checks. Moreover, additional information for further steps can be calculated and attached to the syntax tree.

The semantic checks are performed on the basis of the syntax tree and the attributes of the tokens. Examples for semantic checks are:

- check of variable and function declarations before their use,
- check of static index ranges and
- static type checking.

Semantic checks can be performed interleaved with the syntax analysis or in a separate step through processing the generated syntax tree.

2.3 Compiler Generators

Compiler generators are used to generate parts of compilers. Usually, compiler generators are used to generate scanners and parsers from abstract compiler descriptions. In the following sections the relevant representatives for this work are explained.

2.3.1 *flex / bison*

The scanner generator *lex* [LS75] and the parser generator *yacc* (*yet another compiler compiler*) [Joh75] were developed in the 1970s at the *AT&T Bell Laboratories*. These generators are supplied as standard tools for UNIX operating systems.

Flex (*fast lexical analyzer generator*) [PEM07] and *GNU bison* [DS11] are the open source variants of *lex* and *yacc*. They add improvements and extensions to the original generators. For compatibility, they can process input files for *lex* and *yacc* with only minor modifications.

Flex generates a deterministic finite automaton (DFA) which performs the recognition. *Bison* creates a table-driven bottom-up parser. This LALR-parser (lookahead-LR-parser) is a modified LR-parser. This parser allows smaller tables and decreases therefore the memory requirements.

Due to these characteristics, *bison* can process S/LR-attributed LALR(1) grammars. *Flex* and *bison* generate C/C++ scanners and parsers. A good introduction into *flex* and *bison* gives [Lev09].

2.3.2 *Coco-2*

Coco-2 is a compiler generator which can process L-attributed LL(1) grammars. *Coco-2* was developed with the aim that only one file for the description of the scanner and parser is needed. For this compiler description *Coco-2* uses the input language *Cocol-2* (see section 2.4.2).

Additionally a new top-down parsing method was introduced. The resulting compilers handle the syntax analysis with a table driven top-down parser and evaluate the semantic actions as in a recursive descent parser.

The compiler generator consists of a scanner generator (*SGT*) and a parser generator (*PGT*) which read the same input file. Both generators use as basis a *Top-down-Graph* (*TdG*) which enables checks like the LL(1) condition. The *SGT* generates a minimized deterministic finite automaton. The *PGT* generates a syntax analyzer and a semantic evaluator. The syntax analyzer

is a virtual machine, which interprets a special bytecode (grammar code – Gcode).

Compilers generated by *Coco-2* are efficient with respect to storage and run-time. *Coco-2* is characterized by the simple and powerful input language *Cocol-2* and is available for many programming languages (in and for Pascal, Modula-2, Ada, C, Java, ...). A detailed introduction into *Coco-2* give [DP90] and [DP91].

2.3.3 *Smart*

Smart was developed in 1985 at the Johannes Kepler University in Linz. *Smart* is a compiler generator which can handle L-attributed LALR(1) grammars. *Smart* uses as input language *Cocol* which is characterized by its simplicity and expressiveness.

Compilers generated by *Smart* introduced a new two-step strategy to perform the syntax and semantic analysis. This strategy allows the use of the best method in each step. The first step is performed by a bottom-up LALR(1) analyzer which uses a table-driven approach. The second step is performed by a top-down semantic evaluator.

In the first step the generated compiler produces an intermediate representation (IR) of the source program and a reduction sequence which represent the visited rules. This IR is saved to a file. The second step reads this file and executes the semantic code.

Compilers generated by *Smart* are characterized by a small memory footprint and can therefore be used by micro computers. *Smart* is only available for the programming language *Modula-2*. A detail description of *Smart* is given in [Mös87].

2.4 Compiler Description Languages

Compiler description languages are based on the concepts of formal languages. [Dob11] describes the theory of formal languages as follows:

> Formal languages deal with the structural design, so the syntactical form of ("character") "chains" or better "symbol strings".

In contrast to natural languages which are composed of syntax, semantics and phonetics, a formal language consists of syntax only. A formal grammar consists of rewriting rules. This rewriting rules have non-terminal symbols

Table 2.2: Comparison of grammar notation BNF and Wirth'sche EBNF

Usage	Notation	
	BNF	Wirth'sche EBNF
terminal	words with meaning	words enclosed with "..."
non-terminal	words in <...>	words (large or small initials)
definition	::=	=
alternation	|	|
option		[...]
repetition		{...}
grouping		(...)
termination		.

(set of symbols which occur on both sides of the rules) and terminal symbols (set of symbols which occur only on the right side of the rules). The rules can be read as: "The left side can be replaced by the right side".

Over time, different grammar notations have been developed. For this work BNF and EBNF are essential. BNF was initially invented to describe the syntax of *ALGOL60* [Bac+63]. Based on BNF various extensions have been created. Niklaus Wirth developed an extension to EBNF [Wir77] which has now standardized by the International Organization for Standardisation (ISO) [ISO96]. Today, the names Wirth'sche EBNF and EBNF are used as synonyms. The BNF and EBNF are equally powerful, but the new concepts of EBNF lead to more compact and expressive rules. Table 2.2 shows a comparison of the notational elements (meta symbols) of both grammar notations.

Another notation that is commonly used in compiler description languages is the regular expression (regex). Regular expressions are used to describe regular languages. Regular expressions were first described by [Kle56] and first used in informatics by the text editor *QED* [KRT72] for pattern matching and text replacement. The table 2.3 shows some basic notations of regular expressions used by *flex*.

2.4.1 *flex* and *bison*-Language

Figure 2.2 shows the overall structure of *flex* and *bison* files.

Both files have a definitions section. This is the place for definitions used by later sections. The definition section consists of two parts: The first part contains source code which is copied into the generated scanner and parser.

Table 2.3: Basic notations of regular expressions (taken from [PEM07])

Regex	Description
x	match the character 'x'
.	any character (byte) except newline
rs	the regex 'r' followed by the regex 's'
r|s	either an 'r' or an 's'
(r)	match an 'r', paranthesis are used to override precedence
r*	zero or more r's
r+	one ore more r's
r?	zero ore one r
[xyz]	a character class; in this case, the pattern matches either an 'x', a 'y', or a 'z'
[a-z]	a character class with a range in it, any letter from 'a' to 'z'
{name}	the expansion of the name definition

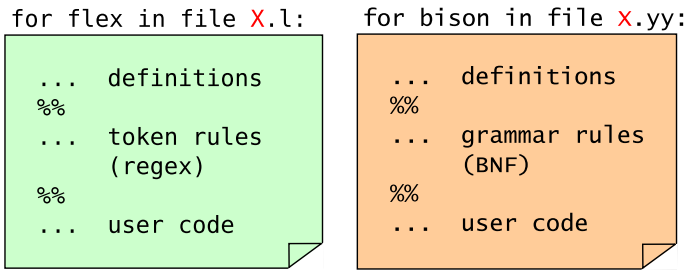


Figure 2.2: Structure of *flex* and *bison* inputs

This is the place to define macros or to declare variables and functions. The second part contains declarations for *flex* and *bison*. For *flex* these are start conditions, options and reusable name definitions. For *bison* this section contains options and declarations of terminal and non-terminal symbols.

The rules section of *flex* contains rules of the form: `pattern action`. The pattern is a regular expression which matches lexemes and executes the corresponding action. The rules section of *bison* contains grammar rules written in BNF-notation. These rules can additionally contain semantic actions.

Finally, the user code section is the place to put user defined code.

A detailed description of the *flex* language gives the *flex manual* [PEM07]. The *bison* language is explained in detail in the *bison manual* [DS11].

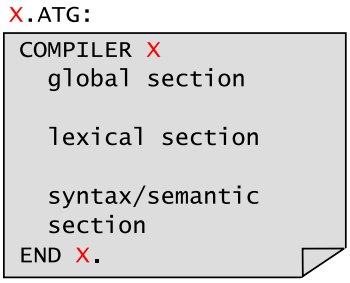


Figure 2.3: Structure of *Coco-2* inputs

2.4.2 *Coco* Language 2 (*Cocol-2*)

Figure 2.3 shows the structure of *Cocol-2* files. In *Cocol-2*, a scanner and parser is defined within one file. This file contains sections to describe the different parts of the compiler. All of these sections are optional. In *Cocol-2*, the scanner and parser definitions are described by the powerful EBNF.

The *Cocol-2* file starts with the definition of the compiler/s name. The next section is the global section. This section contains two source code areas, the first one is copied to the resulting scanner and the second one to the resulting parser.

The next section is the lexical section. This section describes the parts of the scanner. These are the character sets, comments, keywords, tokens, token classes and pragmas.

The syntax and semantics section contains the parts which describe the parser and semantic evaluator. These are the non-terminals and the grammar rules. The grammar rules can contain freely positioned semantic actions which are specified in the target language.

A detailed specification of *Cocol-2* can be found in [DP90].

2.5 Summary

This chapter introduced those areas of compiler construction which have a relevance for this work. At the beginning, ATGs and their classification have been presented in detail. The next section described the structure of a compiler front-end. After that, existing compiler generators and their corresponding compiler description languages have been explained.

The next chapter presents the design of the preprocessor *BoB*. In particular, the design decisions are discussed in detail.

Chapter 3
Design

This chapter covers the design of *BoB*. First, the general solution approach is outlined. After that, the compiler description language *Cocol4BoB* is presented. The next section describes the architecture of *BoB*. The last section describes the architecture of compilers produced by *BoB*.

3.1 Solution Approach

Figure 3.1 shows a comparison of the power of the LL(k) and the LR(k) analysis (see section 2.1). From this figure, it is evident that the top-down analysis has less information during processing the input. In the bottom-up analysis, the symbols already pushed onto the stack are available additionally.

Another important aspect is, that the top-down analysis has access to all semantic attributes that have already been processed. On the other side, the bottom up analysis can only process synthesized attributes, due to the lack of information. [Aho+06] gives the following key statement on the question "Can We Handle L-Attributed Syntax Driven Definition's on LR Grammars?"

> Build the parse tree first and then perform the translation.

This leads to the design decision that syntactic analysis must be performed in bottom-up manner and semantic analysis in top-down manner.

Figure 3.2 shows a general overview of the solution approach. *BoB* processes an ATG written in *Cocol4BoB* and the predefined frame files. *BoB* delivers the input-files for *flex* and *bison*, which are needed to generate the scanner and parser. This intermediate step has the advantage that *BoB* does not have to generate a complex bottom-up parser, because *bison* already provides a well-tested one. Additionally *BoB* delivers a semantic analyzer, which is compatible with the scanner and parser generated by *flex* and *bison*.

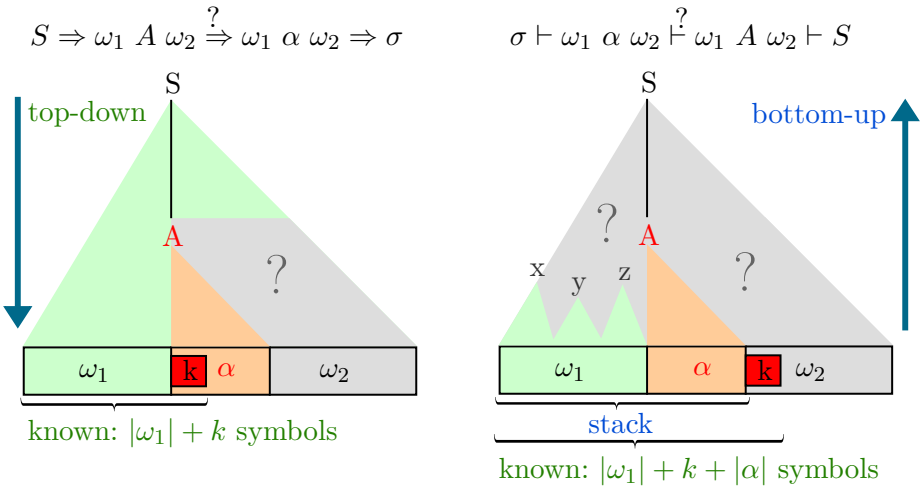


Figure 3.1: Power – LL(k) versus LR(k) [Dob11]

The generated compiler processes the input files in two steps. The first step is lexical analysis and bottom-up syntax analysis. The second step is the top-down semantic analysis. These two steps must be strictly separated. For information exchange, appropriate data structures are used, that store the lexical and syntactical information. In case of syntactical errors, the second step can be skipped.

Another important aspect of the design is that *BoB* itself is generated by *flex* and *bison*. This brings the possibility to perform a bootstrap. Bootstrapping enables the developer to test whether the preprocessor generated by the programmer functionally matches the preprocessor generated by *BoB* (see section 5.3).

3.2 Specification of *Coco/4BoB*

An important design decision concerns the definition of an adequate input language. For *BoB* a new compiler description language *Coco/4BoB* was developed. For *Coco/4BoB* the following criteria were formulated:

user friendly – This means, less constructs to describe the parts of the compiler (e.g., only EBNF for scanner and parser instead of regular expressions and BNF).

single source – Only one input file, that ensures consistency and avoids multiple declarations.

Due to these requirements, the compiler description language *Coco/4BoB* can be similar to *Coco-2* (see section 2.4.2).

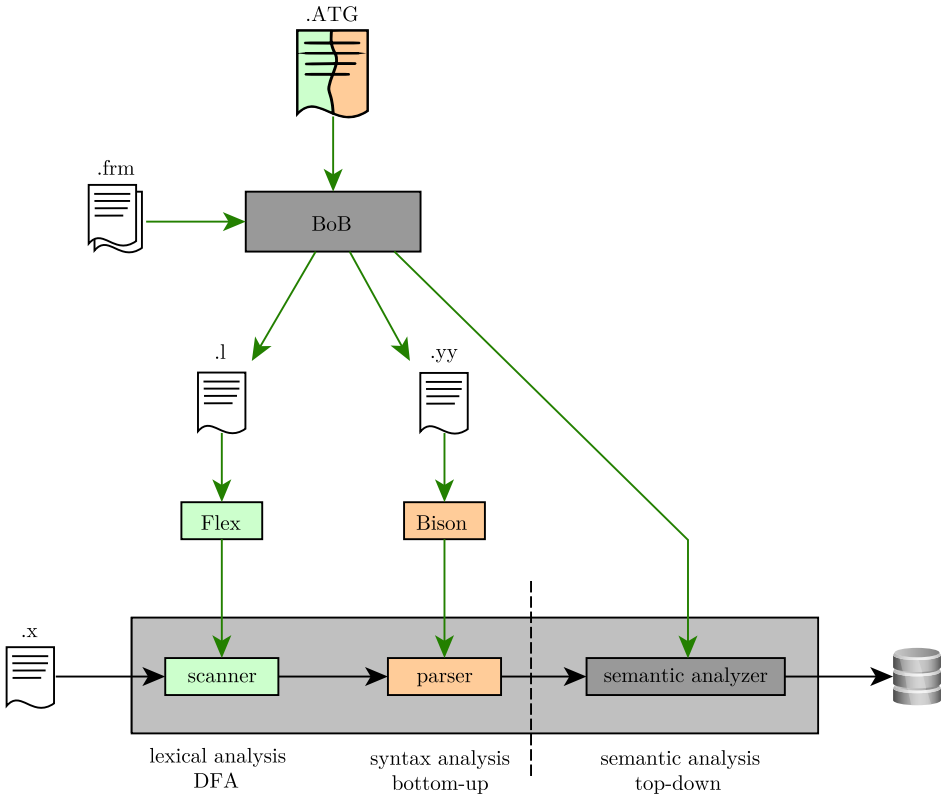


Figure 3.2: General overview

3.2.1 Changes to *Cocol-2*

Cocol-2 can not be used without any changes, since it uses reserved tokens from the target language C++. Moreover, the targets (*flex* and *bison*) do not support all concepts of *Cocol-2*. Therefore, *BoB* uses a modified version of *Cocol-2* with the name *Cocol4BoB*. The following list shows the changes and restrictions that were applied:

- `PRAGMA`s in *Cocol-2* support semantic actions. This is not possible in *flex* and *bison*. The *bison* parsers do not allow tokens that occur at any position in the source code.
- In *Cocol-2* the source code section is defined as `PRAGMA`. As described above, `PRAGMA`s do not allow semantic actions. Because the source code handling uses semantic actions, the source code section has to be redefined within the token-class section. The old and the new concepts are interchangeable and the new solution has no negative effects.
- The start and end-delimiters for source code within *Cocol-2* are identical

with the input- and output operators of C++. Therefore, the operators `<<` and `>>` are replaced by `<|` and `|>`.

- In *Cocol-2* you can specify an alias for a non-terminal, for meaningful error messages. *Bison* do not support such aliases. Therefore they are removed. Exactly the same is true for the token-declaration. For the alternative, which takes a string as token description, no alias is allowed. *Flex/Bison* does not support this.
- The generated semantic evaluator is a C++-class, which is defined within a header and an implementation file. Because the developer should have the possibility to extend both files, a new section called `SEMDECL` is introduced. This section holds the user code copied into the header file.

3.2.2 Small Example

The following example is a complete input file written in *Cocol4BoB* for a simple compiler *Calc* which evaluates arithmetic expressions. *Calc* checks the input for lexical and syntactical correctness and calculates the result in the semantic actions. Finally, the result is written to the console. An advanced example which describes the compiler *BoB* can be found in Apendix A.

Cocol4BoB

```
1  COMPILER Calc
2
3  CHARACTER SETS
4    digit = '0' .. '9'.
5    whiteSpace = CHR(9) + EOL IGNORE.
6                 /*' ' ignored by default*/
7
8  COMMENTS
9      FROM '--' TO EOL.           --Ada comments
10     FROM '//' TO EOL.           //C++ comments
11     FROM '/*' TO '*/'.          /*C comments*/
12     FROM '(*' TO '*)' NESTED.   (*Modula-2 comments*)
13
14 TOKENS
15   '+'. '-'.
16   '*'. '/'.
17   '('. ')'.
18
19 TOKEN CLASSES
20   number<|int& val|> =
21     digit { digit }             LEX <|val = atoi(tokenStr);|>.
22
23 NONTERMINALS
24   Calc.
```

```
25   Expr<|int &e|>.
26   Term<|int &f|>.
27   Fact<|int &t|>.
28
29 RULES
30 Calc =                          LOCAL<|int e = 0;|>
31   Expr<|e|>                     SEM<|cout << "res: " << e;|>.
32
33 Expr<|int &e|> =                LOCAL<|int t = 0;|>
34   Term<|e|>
35   { '+' Term<|t|>               SEM<|e = e + t;|>
36   | '-' Term<|t|>               SEM<|e = e - t;|>
37   }.
38
39 Term<|int &t|> =                LOCAL<|int f = 0;|>
40   Fact<|t|>
41   { '*' Fact<|f|>               SEM<|t = t * f;|>
42   | '/' Fact<|f|>               SEM<|t = t / f;|>
43   }.
44
45 Fact<|int &f|> =
46      number<|f|>
47   | '(' Expr<|f|> ')'.
48
49 END Calc.
```

3.3 Architecture of *BoB*

The class diagram in Figure 3.3 shows the general architecture of *BoB*. *BoB* consists of one executable *BoB* and two static libraries *BoBCommon* and *BoBTemplate*. The library *BoBCommon* contains the common parts of *BoB* and the compilers generated from *BoB*. The library *BoBTemplate* encapsulates all parts which are related to the template mechanism used by *BoB*.

The executable *BoB* has a class *BoB* which is the main entry point into the preprocessor. This class is responsible for handling user input/output and for initiating the scanner and the parser as well as the semantic analyser. All other classes in this executable are generated from the *flex* and *bison* input files. The classes *Location* and *Position* are used by the parser for location tracking. The class *Stack* is a custom stack implementation generated by *bison*, which is used by the parser. The class *BoBScanner* and *BoBParser* represent the scanner and parser.

The library *BoBCommon* provides utility classes. The class *Errors* provides the error tracking mechanism. An object of this class stores the error messages

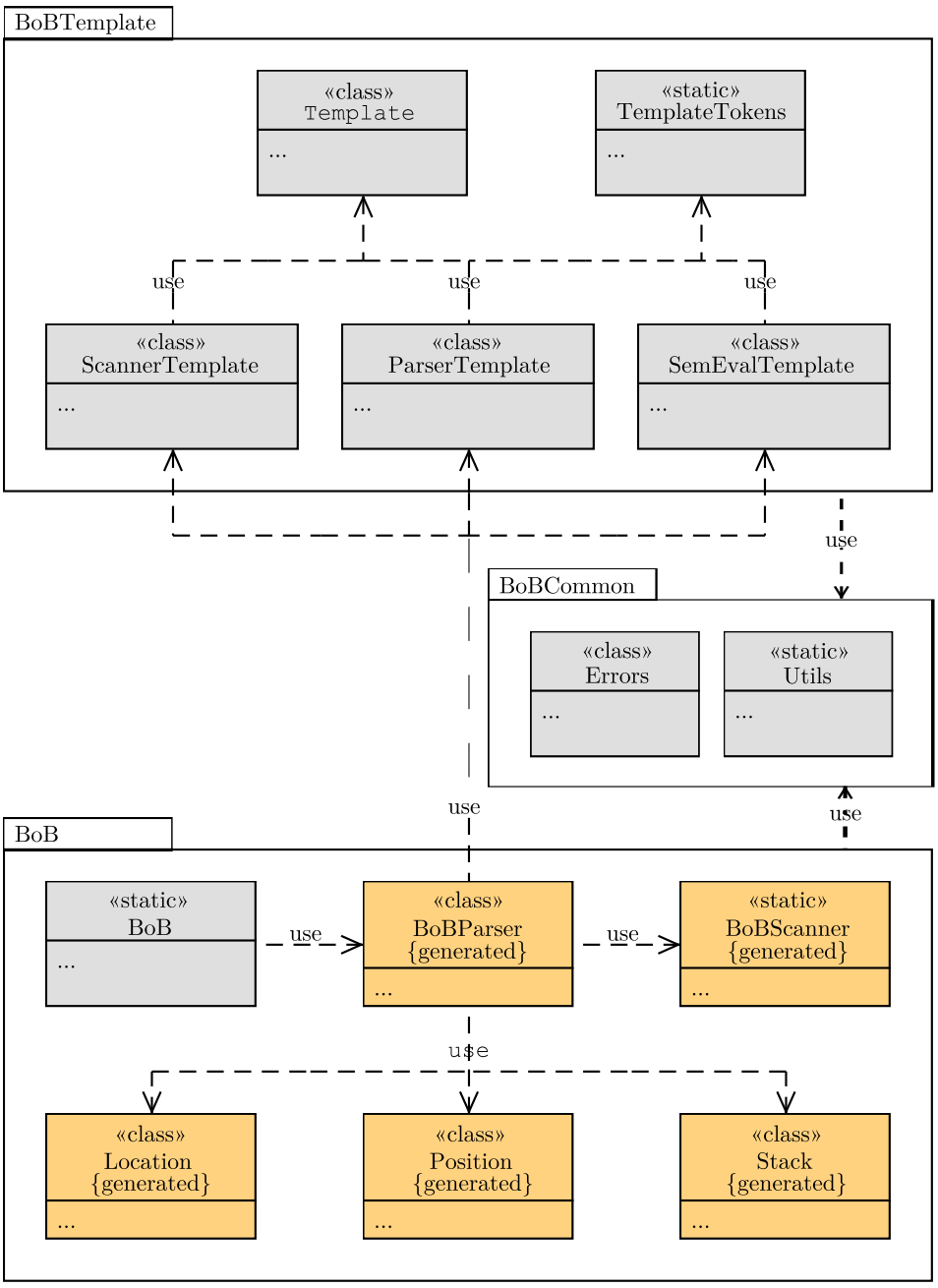


Figure 3.3: Architecture of BoB

collected during parsing. Moreover, it provides a mechanism for reporting the errors to the user. The second class *Utils* provides static methods for common tasks:

- a timer functionality for measuring execution times,
- path utilities for handling file system operations and
- string utilities for commonly used string operations, like escaping input characters for the generated regular expressions.

The library *BoBTemplate* is the centerpiece of *BoB*. This library contains the complete transformation algorithms used for the transformation from *Cocol4BoB* to *flex/bison*. The central part of this library is the class *Template*. This class provides a key-value store which holds the assignments of the placeholders to the target content. Based on this key-value store, it provides the possibility to fill a frame which contains the corresponding placeholders. The class *TemplateTokens* provides the definitions of the used frames and the used placeholders.

The list below explains the provided frames:

- **Main.frm** – This frame is a default implementation of the main file of the compiler. This frame is not used by *BoB*, but it can used by the developer to create a basic compiler.
- **Scanner.frm** – The scanner frame holds the constant parts of the generated *flex* file.
- **Line-, Block- and NestedBlockComment.frm** – These frames are used to build the *flex* comments for the scanner frame.
- **Parser.frm** – The parser frame holds the constant parts of the generated *bison* file.
- **SemEvalH.frm, SemEvalCpp.frm** – These frames hold the constant parts of the header and the implementation file of the semantic evaluator.
- **SemMethod.frm** – This frame is used to build a semantic method.

The explained frames are used by the classes *ScannerTemplate*, *ParserTemplate* and *SemEvalTemplate*. These classes differ in the particular target component. They contain the transformation methods, which take as input the data provided by the scanner and parser. These methods create the desired outputs and fill the associated key-value store. Finally, after the whole input processing, the output files are generated by filling the frames. Figure 3.4 shows the basic concept of filling the frames. The placeholders within the frames can occur multiple times. The placeholders must be unique and as short as possible. Therefore, the pattern `==>[Identifier][Number]` is used. Where the identifier stands for a specific file and the number is an ongoing number inside the file.

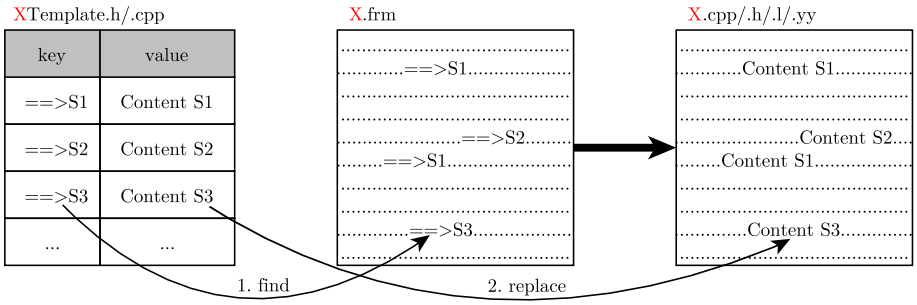


Figure 3.4: Filling the frames

3.4 Architecture of the Compilers generated by *BoB*

The class diagram in Figure 3.5 shows the architecture of compilers generated by *BoB*. The X in the diagram stands for the name of the generated compiler. The main class *X* can be derived from the provided *Main.frm* or implemented by the developer. The *BoBCommon* library is identical to the library used by *BoB* as described in the previous section. It contains the error handling and some utility functionality.

The classes *XParser*, *XScanner*, *Location*, *Position* and *Stack* are generated by the tool-chain *BoB-flex-bison*. These classes have exactly the same functionality as within *BoB*. The new class *XSemEval* contains the semantic evaluator. This class contains the intermediate data structures, which are filled by the scanner and parser. Additionally this class contains semantic methods for each rule of the input ATG. This is different to *Smart* which uses special semantic code. The semantic evaluation is initiated by the main class *X*.

3.5 Summary

This chapter gave an overview of the design of *BoB*. First, the general approach was introduced. This was then concretized in the following sections. After that, the newly introduced compiler description language *Cocol4BoB* was presented. Finally, the chosen architecture of *BoB* and of the compilers generated by *BoB* were explained.

The next chapter describes the implementation of the described design.

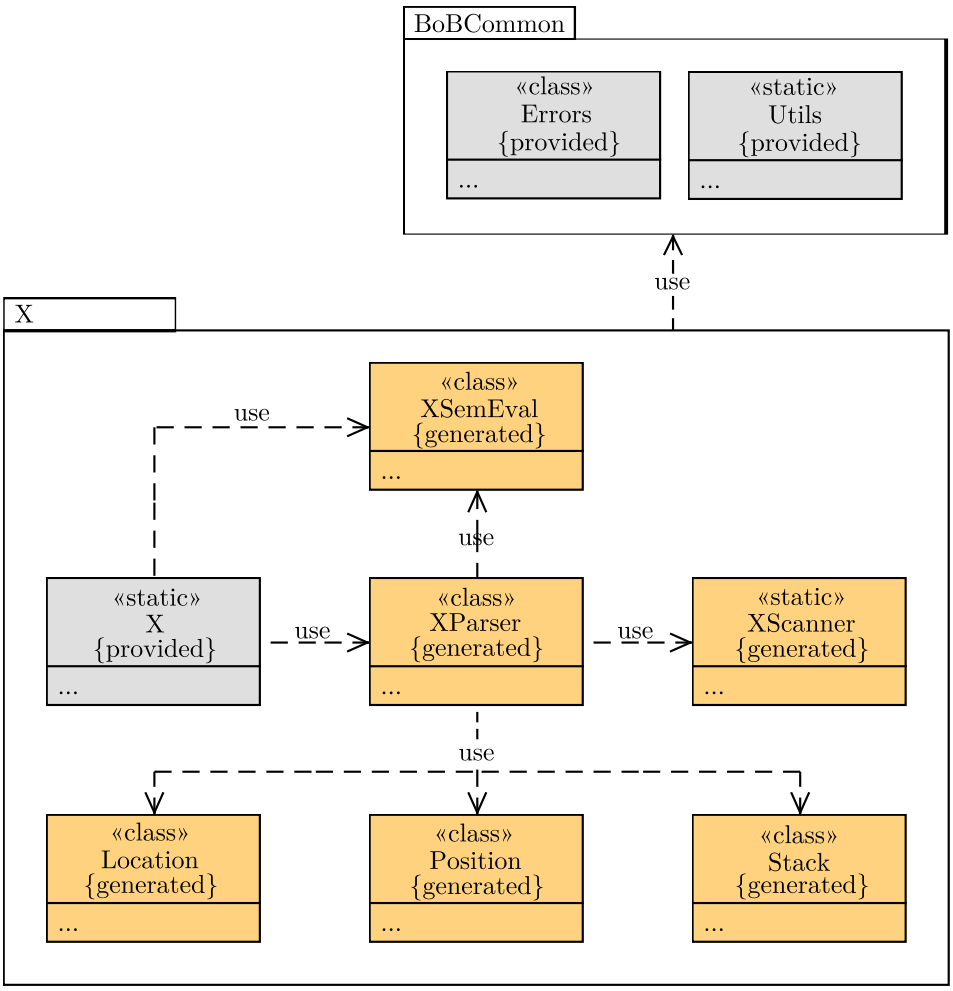


Figure 3.5: Architecture of compilers generated by *BoB*

Chapter 4
Implementation

The previous chapter presented the design of *BoB*. This design serves as a basis for the implementation, which will be explained in detail in this chapter. It is divided in three sections corresponding to the main components of *BoB*. The first section explains the relevant scanner parts. The next section describes the generation of the parser. The third section describes the semantic evaluator.

4.1 Transformation from *Cocol4BoB* to *flex* Input

BoB gets as input an ATG which describes the compiler to be generated. *BoB* extracts the relevant parts for the scanner and uses a template mechanism to fill in a predefined frame. Figure 4.1 shows the relevant parts of the input file and the overall structure of the generated output file.

In the following subsections, the individual parts are explained in detail. First, the frame with the constant parts are described. After this, the character sets, comments and token classes are presented in detail as examples for the used concepts. Finally, the remaining elements are explained.

4.1.1 Basic Frame for *flex* Input

The basic frame of the output consists of the constant parts of the output file. These parts are located in the definition section of the *flex* input file. In the first section, the includes for the parser and semantic evaluator are defined. These includes are needed to publicize the parser tokens to the scanner and to enable the exchange of the lexical data from the scanner to the semantic evaluator.

The next section holds the function prototypes and the declaration of the signature of the main *lex* function. *Flex* provides for this a macro `YYDECL`

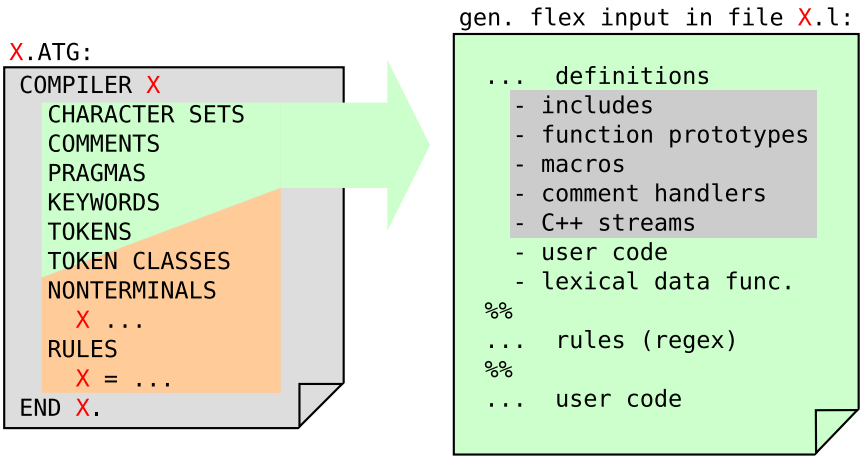


Figure 4.1: General overview of the transformation from *Cocol4Bob* to *flex*

which can be changed. The following code snipped shows the implementation for *BoB*. *Flex* and *bison* define their data types within the namespace *yy*. The sequence `==>S1` is a placeholder for the compiler name.

C++
```
#define YY_DECL                                              \
      yy::==>S1Parser::token_type                            \
      yylex (yy::==>S1Parser::semantic_type* yylval,  \
             yy::==>S1Parser::location_type* yylloc)
```

Additionally to that, there are utility macros for tracking the line and column numbers. This information is used in error messages to provide adequate error positions. The `COLUMNS` macro increases the column location. The `LINES` macro increases the line location. The macro `YY_USER_ACTION` is a predefined macro of *flex* which is called after a regular expression match. This is an ideal point to increase the column counter. Therefore, it is not necessary, to increase the column counter within each regular expression. Custom line and column tracking is required only in custom input processing. For example within source code processing.

C++
```
#define COLUMNS do { yylloc->step();  \
            yylloc->columns(yyleng);  \
          } while (false)

#define LINES do { yylloc->step();    \
            yylloc->lines(yyleng);    \
          } while (false)

#define YY_USER_ACTION do {           \
```

```
10              if (YY_START == INITIAL)  \
11                COLUMNS;                \
12            } while(false);
```

The generated *flex* scanner is a C-style scanner. This means, it uses a procedural programming model. For better compatibility to the object-oriented design of the other parts, the `FILE*` type is replaced by C++ streams. For this *flex* provides the macro `YY_INPUT` which can be changed. The streams are predefined to the standard input and output streams. They can be changed by the user. The following source code example shows the implementation of `YY_INPUT`:

C++

```
1  // defined in scanner:
2  #define YY_INPUT(buf, result, max_size)                    \
3    if ((result = LexerInput((char *) buf, max_size)) < 0)  \
4      YY_FATAL_ERROR("input in flex scanner failed");
5
6  static int LexerInput(char* buf, int max_size) {
7    if (yyInStream->eof() || yyInStream->fail()) {
8      return 0;
9    } // if
10   yyInStream->read(buf, max_size);
11   if (yyInStream->bad()) {
12     return -1;
13   } else {
14     return yyInStream->gcount();
15   } // if
16 } // LexerInput
17
18 // defined in parser:
19 extern std::istream* yyInStream;
20 extern std::ostream* yyOutStream;
21
22 std::istream* yyInStream  = &std::cin;
23 std::ostream* yyOutStream = &std::cout;
```

Finally, there are the constant parts for comment treatment. Since the comments are treated in a separate section, this part is described later (see section 4.1.3).

4.1.2 Character Sets

Character sets provide a mechanism to reuse a collection of characters through an identifier. In *Coco/4BoB*, the character sets are built from sets defined by ranges based on the character encoding ASCII. The set operations union and difference are used to build complex sets. Moreover, there are

Table 4.1: Transformation of character set elements

Cocol4BoB	*flex* (Regex)
Special .. Special	[Special-Special]
ANY	[^\n]
EOL	[\n]
CHR(0 ...127)	\x00, ..., \x7f
+	{+}
−	{−}

predefined sets, e.g., `ANY` which holds all characters except the end of line (`EOL`) character. Non-printable characters can be defined with the special construct `CHR(0 ... 127)`. The source code below shows the definition of character sets with the described constructs:

Cocol4BoB

```
CHARACTER SETS
  letter      = 'A'..'Z' + 'a'..'z'.
  digit       = '0'..'9'.
  control     = CHR(0) .. CHR(31) + CHR(127).
  noQuote1    = ANY - "'" - control. (*valid chars in '...'*)
  noQuote2    = ANY - '"' - control. (*valid chars in "..."*)
  ignore      = EOL + CHR(9) IGNORE.
```

The *bison* rules, which process the character sets, are shown below. The methods in the semantic actions build the resulting regular expression for *flex* on the fly. This means, each input leads to a call of a semantic method, which performs the transformation in place. As can be seen from Table 4.1, there exist a equivalent transformations from *Cocol4BoB* constructs to *flex* constructs.

Bison

```
OptSetDecl:
    /* eps */
  | CHARACTER SETS SetDeclBlock;

SetDeclBlock:
    /* eps */
  | SetDeclBlock SetDecl;

SetDecl:
    IDENT '=' ChrSet
    ChrSetBlock OptIgnore '.' { /* SEM: AddSetDecl        */ }
  | error '.';

ChrSet:
```

```
15      IDENT                       { /* SetDeclAddIdent        */ }
16    | ANY                         { /* SetDeclAddANY          */ }
17    | EOL                         { /* SetDeclAddEOL          */ }
18    | Special OptSpecial;
19
20  Special:
21      STRING                      { /* SetDeclAddSpecial      */ }
22    | CHR '(' NUMBER ')'          { /* SetDeclAddSpecial      */ };
23
24  OptSpecial:
25      /* eps */                   { /* SetDeclEndSpecial      */ }
26    | ".."                        { /* SetDeclAddRange        */ }
27      Special                     { /* SetDeclEndSpecial      */ };
28
29  ChrSetBlock:
30      /* eps */
31    | ChrSetBlock PlusOrMinus ChrSet;
32
33  PlusOrMinus:
34      '+'                         { /* SetDeclSetSign - PLUS  */ }
35    | '-'                         { /* SetDeclSetSign - MINUS */ };
36
37  OptIgnore:
38      /* eps */                   { /* SetDeclSetIgnore false */ }
39    | IGNORE                      { /* SetDeclSetIgnore true  */ };
```

The transformation is straightforward. However, there are two exceptions:

1. The first refers to the reuse of named character sets within another character set, like in the *noQuote* examples. *Flex* basically supports the use of named character sets, but has a problem in combination with the set operations. Due to that, it is necessary to replace the identifier with its definition. For this, the corresponding definition of each identifier has to be saved. An important point is, that it is not a simple text replacement, because the rules of set operations have to be considered. So, if the identifier follows a `"-"` (difference) the signs within the implementation must be turned.
2. The second exception is the optional keyword `IGNORE`. This keyword indicates that character sets should be ignored by the scanner. In *flex* this is implemented by additional regular expressions with empty actions.

Figure 4.2 shows the semantic methods provided by the singleton class *ScannerTemplate*. As described the methods *SetDeclAdd...* perform the transformation. The method *AddSetDecl* finally adds the constructed *flex* expression to the scanner template.

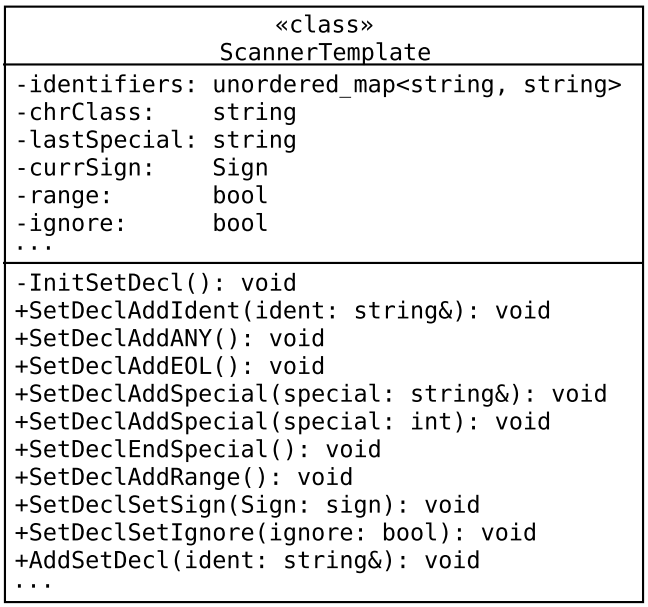


Figure 4.2: *ScannerTemplate* – Methods for the treatment of character sets

The code example below shows the output of the transformation of the input described on page 28. The replacement of the named character sets with the definition is highlighted. Additionally, it can be seen how the signs were inverted.

Flex

```
1  letter    [A-Z]{+}[a-z]
2  digit     [0-9]
3  control   [\x00-\x1f]{+}[\x7f]
4  noQuote1  [^\n]{-}[']{-}[\x00-\x1f]{-}[\x7f]
5  noQuote2  [^\n]{-}["]{-}[\x00-\x1f]{-}[\x7f]
6  ignore    [\n]{+}[\t]
```

4.1.3 Comments

BoB supports three types of comments:

1. The first type is the line comment. This one starts with a start delimiter and ends with the end-of-line (EOL) delimiter.
2. The second type is the block comment. The block comment starts and ends with custom delimiters and ignores line breaks.
3. The last comment type is the nested block comment. It is a block comment, which handles subsequent start and end delimiters. Nested block

comments allow commenting out portions of source code irrespective of existing comments.

In the input language *Cocol4BoB* the comments are defined by a start and end delimiter. This delimiters are defined as strings, identifiers, characters by ASCII code or the special EOL delimiter. The delimiter must not exceed a length of two characters. Optionally, the nesting and a comment handler can be enabled. The following code example shows an example of the definition of the described comment types:

Cocol4BoB

```
COMMENTS
  FROM '//' TO EOL         HANDLER CmntHandler. // C++
  FROM '/*' TO '*/'        HANDLER CmntHandler. /* C */
  FROM '(*' TO '*)' NESTED HANDLER CmntHandler. (* Modula-2 *)
```

The syntactic and semantic processing of the input is defined by following *bison* rules:

Bison

```
OptCommentDecl:
    /* eps */
  | COMMENTS CommentDeclBlock;

CommentDeclBlock:
    /* eps */
  | CommentDeclBlock CommentDecl;

CommentDecl:
    FROM Delimiter            { /* SEM: CmntFromEnd       */ }
    TO Delimiter OptNested
    OptHandler '.'            { /* SEM: AddCmnt           */ }
  | error '.';

Delimiter:
    DelimiterPart
  | DelimiterPart DelimiterPart;

DelimiterPart:
    STRING                    { /* SEM: CmntDelimAddStr   */ }
  | IDENT                     { /* SEM: CmntDelimAddIdent */ }
  | CHR '(' NUMBER ')'        { /* SEM: CmntDelimAddChr   */ }
  | EOL                       { /* SEM: CmntDelimAddEOL   */ };

OptNested:
    /* eps */
  | NESTED                    { /* SEM: CmntSetNested     */ };

OptHandler:
    /* eps */
```

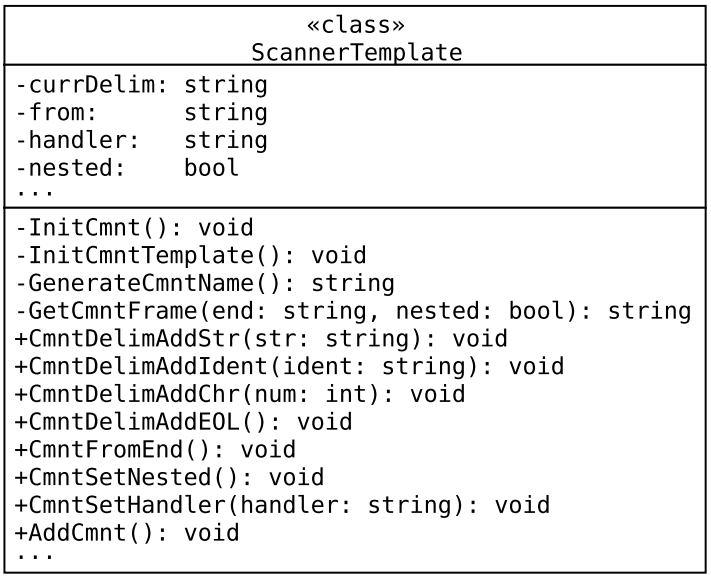


Figure 4.3: *ScannerTemplate* – Methods for the treatment of comments

```
31   | HANDLER IDENT                { /* SEM: CmntSetHandler    */ };
```

The parser, generated by the *bison* rules, checks the syntactic correctness of the input and calls the semantic methods in their semantic actions. The rules *DelimiterPart*, *OptNested* and *OptHandler* collect the input data by calling the methods *CmntDelimAdd...*, *CmntSetNested* and *CmntSetHandler* in their semantic actions. The rule *CommentDecl* finally calls the method *AddCmnt*. This method uses the collected data to fill in a predefined frame. The described semantic methods are provided by the singleton class *ScannerTemplate* (see Figure 4.3).

For each comment type, there exists a different frame. These frames contain the invariant parts of the output. The following code example shows a filled frame for a single **line comment**. The inserted data is highlighted.

Flex

```
1 /* FROM '//' TO EOL HANDLER CmntHandler. */
2 %x LINE_CMNT
3
4 "//"          { BEGIN(LINE_CMNT); ResetCommentHandler(); }
5
6 <LINE_CMNT>{
7   [^\n]+      { DefaultCmntHandler(CmntHandler); }
8   [\n]        { BEGIN(INITIAL); }
9 }
```

The frame consists of three parts:

1. First, the state definition at line 2 defines a *flex* state with the name `LINE_CMNT`.
2. The regular expression in line 4 defines the state entry point. It matches the start delimiter of the comment and uses the `BEGIN(LINE_CMNT)` macro. Generally, the macro `BEGIN(...)` switches between defined states by passing the target state as parameter.
3. After entering the state, *flex* only recognizes the regular expressions defined in the state implementation (lines 6-9). The line comment state defines two regular expressions:
   (a) The first consumes all the symbols within the comment and optionally handles them with a comment handler.
   (b) The second regular expression matches the EOL symbol and leaves the state by calling the macro `BEGIN(INITIAL)`. The state `INITIAL` is the predefined global state of *flex*.

The next code example shows the filled frame for a **block comment**. The basic concepts are identical to the single line comment. But the regular expressions within the state are different.

1. The first expression catches the case of a possible end of file (EOF) within a block comment.
2. The next expression consumes all symbols with no special meaning.
3. The expression `"*"` matches the occurrence of the first character of the end delimiter.
4. The expression which handles the EOL symbol is necessary for the calculation of the line numbers which are needed by error messages.
5. Finally the last expression matches the end delimiter which leads to leave the state. This works, because *flex* always tries the longest match.

Flex

```
1 (* FROM '/*' TO '*/' HANDLER CmntHandler. *)
2 %x BLOCK_CMNT
3
4 "/*"          { BEGIN(BLOCK_CMNT); ResetCommentHandler(); }
5
6 <BLOCK_CMNT>{
7   <<EOF>>     { (* Error EOF within comment *)   }
8   [^*\n]+     { DefaultCmntHandler(CmntHandler); }
9   "*"         { DefaultCmntHandler(CmntHandler); }
10  [\n]+       { DefaultCmntHandler(CmntHandler); }
11  "*/"        { BEGIN(INITIAL); }
12 }
```

The last code example shows the filled frame of the **nested block comment**. The frame for the nested block comment has the same structure as the frame

of the block comment. As extension, there is a new regular expression which handles the occurrence of the start delimiter within the comment. In the semantic action this rule increases a depth counter. The rule which matches the end delimiter leaves the state only at depth 0. Otherwise it decreases the depth counter only. The first character of the delimiters have to be handled separately with own rules (see line 13 and 14).

Flex

```
1  /* FROM '(*' TO '*)' NESTED HANDLER CmntHandler. */
2  %x NESTING_COMMENT
3
4  "(*"          { BEGIN(NESTING_COMMENT);
5                    ++depth;
6                    ResetCmntHandler();
7                }
8
9  <NESTING_COMMENT>{
10   <<EOF>>     { /* Error EOF within comment */ }
11   "(*"        { ++depth; DefaultCmntHandler(CmntHandler); }
12   [^(*\n]+    { DefaultCmntHandler(CmntHandler); }
13   "("         { DefaultCmntHandler(CmntHandler); }
14   "*"         { DefaultCmntHandler(CmntHandler); }
15   [\n]+       { DefaultCmntHandler(CmntHandler); }
16   "*)"        { --depth;
17                 if (depth == 0)
18                   BEGIN(INITIAL);
19                 else
20                   DefaultCmntHandler(CmntHandler);
21               }
22 }
```

For each comment, a comment handler can be registered. A comment handler is defined by its name and a corresponding function. The parameters (the comment text, the character count and the level) of the function are predefined. The utility function *ResetCommentHandler* initializes the comment handler. The utility function *DefaultCmntHandler* calculates the parameters and calls the comment handler with the provided function pointer. The implementation of the utility functions can be seen in the source code below:

C++

```
1  static int  depth = 0;
2  static bool handleComment = true;
3  static int  comChCount = 0;
4
5  typedef bool (*CommentHandlerPtr)(char*, int, int);
6
7  static void ResetCommentHandler() {
8    depth = 1;
9    handleComment = true;
```

```
10   comChCount = 0;
11 } // ResetCommentHandler
12
13 static void DefaultCommentHandler(CommentHandlerPtr handler) {
14   if (handleComment) {
15     comChCount += yyleng;
16     handleComment = handler(yytext, comChCount, depth + 1);
17   } // if
18 } // DefaultCommentHandler
19
20 static bool CmntHandler(char* cmntText,
21         int chCount, int level) {
22   /* do something */;
23   return true;
24 } // CmntHandler
```

4.1.4 Token Classes

Token classes are terminal symbols with attached lexical data – like identifiers, strings, numbers and source code (as special case within *BoB*). The declaration of a token class consists of an identifier with a parameter list for the return value for the lexical data. Optionally, an alias can be defined. The definition is a lexical expression which is followed by a lexical declaration and an action. The following example shows the definition of the token class *ident* for identifiers:

Cocol4BoB

```
1 TOKEN CLASSES
2   ident <|string& s|> ALIAS 'identifier' =
3     letter {
4       letter
5     | digit
6     | '_'
7     } LEX <| s = string(tokenStr); |>.
```

The lexical expression is a complex rule, because the rule can contain indirect cycles. Due to that, the transformation has to be processed during parsing like the transformation of the character sets (see section 4.1.2). The bison rules below define the syntactic and semantic processing of the token classes:

Bison

```
1 OptTokenClassDecl:
2     /* eps */
3   | TOKEN CLASSES TokenClassDeclBlock;
4
5 TokenClassDeclBlock:
6     /* eps */
7   | TokenClassDeclBlock TokenClassDecl;
```

```
8
9  TokenClassDecl:
10     IDENT SourceBlock OptAlias '='
11     LexExpr OptLexDecl
12     OptLexAction '.'           { /* SEM: AddTokenClass     */ }
13   | error '.';
14
15 OptAlias:
16     /* eps */
17   | AliasName;
18
19 AliasName:
20   ALIAS IdentOrString;
21
22 IdentOrString:
23     IDENT
24   | STRING;
25
26 LexExpr:
27   LexTerm LexTermBlock;
28
29 LexTerm:
30     LexFact LexFactBlock OptIfFollowedBy;
31
32 LexFact:
33     IDENT                      { /* SEM: AddIdent          */ }
34   | STRING                     { /* SEM: AddString         */ }
35   | CHR '(' NUMBER ')'         { /* SEM: AddChar           */ }
36   | EOL                        { /* SEM: AddSymbol - [\n]  */ }
37   | '('                        { /* SEM: AddSymbol - (     */ }
38     LexExpr ')'                { /* SEM: AddSymbol - )     */ }
39   | '['                        { /* SEM: AddSymbol - (     */ }
40     LexExpr ']'                { /* SEM: AddSymbol - )?    */ }
41   | '{'                        { /* SEM: AddSymbol - (     */ }
42     LexExpr '}'                { /* SEM: AddSymbol - )*    */ }
43   ;
44
45 LexFactBlock:
46     /* eps */
47   | LexFactBlock LexFact;
48
49 OptIfFollowedBy:
50     /* eps */
51   | IF FOLLOWED BY '('         { /* SEM: AddSymbol - /     */ }
52     LexExpr ')';
53
54 LexTermBlock:
55     /* eps */
56   | LexTermBlock '|'           { /* SEM: AddSymbol - |     */ }
```

Table 4.2: Transformation of token class elements

Cocol4BoB	*flex* (Regex)
(...|...)	(...|...)
[...]	(...)?
{...}	(...)*
`IF FOLLOWED BY` (...)	/...

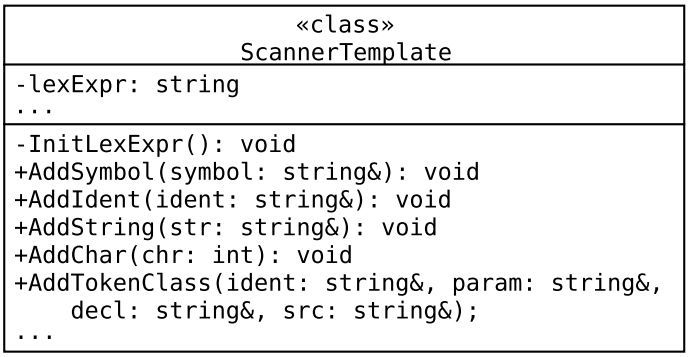


Figure 4.4: *ScannerTemplate* – Methods for the treatment of token classes

```
57      LexTerm;
58
59  OptLexDecl:
60      /* eps */
61    | LEXLOCAL SourceBlock;
62
63  OptLexAction:
64      /* eps */
65    | LEX SourceBlock;
```

Table 4.2 compares the *Cocol4BoB* syntactical constructs with the equivalent *flex* syntactical constructs. Due to the fact that there are equivalent replacements in *flex*, the transformation can be processed in the *Add...* methods. The method *AddSymbol* is a generalized method which takes the new representation as parameter. The method *AddTokenClass* finishes the detection and generates methods for the handling of the lexical data associated with the token class. This part is described in the section 4.3.1. Figure 4.4 shows the methods described in this paragraph.

4.1.5 Other Parts of *Cocol4BoB* related to *flex* Input

The previous three sections described the main concepts of the transformation from *Cocol4BoB* to *flex*. This section presents the remaining components. This are the keywords, tokens and pragmas:

- Keywords are reserved words of the input language of the scanner. In *Cocol4BoB* keywords can defined by an identifier or a string. Both versions allow aliases. For the generated *flex* file, each keyword forms a regular expression which matches the keyword and delivers an associated token to the parser.
- Tokens are usually used for special characters like mathematic operators. It has to be distinguished between tokens matched by an regular expression of the token name and tokens which are matched by an lexical expression. In principle, tokens are treated like keywords. With one exception: the single character tokens. The single character tokens can be handled with a generalized regular expression which recognizes all of them.
- The implementation of pragmas is similar to the structure of the token class implementation. The main difference is, that pragmas are scoped to the scanner. Pragmas can handle the input, but can not interact with the parser. Usually, pragmas are used to define compiler macros like `#DEF`, `#UNDEF` and `#IF`. For the necessary concepts see the token class implementation in section 4.1.4.

4.2 Transformation from *Cocol4BoB* to *bison* Input

The input file for *bison* is also generated from the attributed grammar. Figure 4.5 shows the parts, which are relevant for the parser. It is apparent that *flex* and *bison* use the information of the keywords, tokens and token classes corporately. Nevertheless, to avoid that the ATG has to be read twice, the output frames are filled concurrently.

The following section describes the frame for the generated *bison* file. The next section explains the transformation of the grammar rules. The last section explains the error recovery used by *BoB* and by compilers generated by *BoB*.

4.2.1 Basic Frame for *bison* Input

As with the scanner, the basic frame holds the constant parts of the generated parser file. These constant parts are placed in the definition section of the

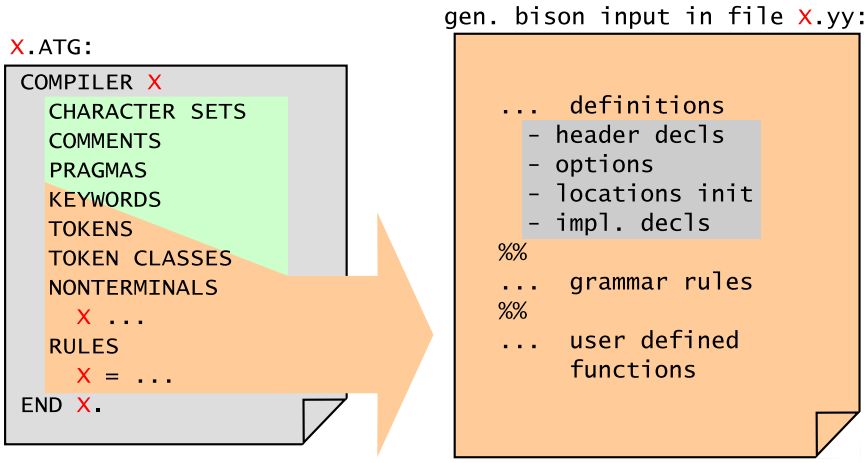


Figure 4.5: General overview of the transformation from *Coco/4Bob* to *bison*

file. A difference to the scanner is that the *bison* file generates a C++ parser. For that, *bison* provides different sections for declarations in the header and in the implementation file.

Within the block `%code requires { ... }` declared includes and declarations are added to the header file of the generated parser. The source code example below shows the includes and declarations which are added to the header file:

C++
```
%code requires {
  #include <string>
  #include <iostream>
  #include <fstream>

  // input and output streams for flex and bison
  extern std::istream* yyInStream;
  extern std::ostream* yyOutStream;
}
```

The next parts of the definition section are the parser options. *Bison* has many options which can be set for parsing a *bison* file. Usually, these options are provided via the command line. For better usability, a pre-selection of the *bison* options was made:

C++
```
%require "2.7"
%skeleton "lalr1.cc"
%define parser_class_name "==>P1Parser"
%locations
%error-verbose
```

```
%debug
%verbose
```

These options define:

- the minimum *bison* version → 2.7,
- the skeleton of the generated *bison* parser → C++,
- the name of the resulting parser,
- that location tracking by *bison* is enabled and finally
- the logging options.

As described above, the location tracking provided by *bison* is used. For this, *bison* provides an initialization block which is called before starting the parser. In this block the locations are reset.

C++

```
%initial-action {
  @$.begin.line     = @$.end.line     = 1;
  @$.begin.column   = @$.end.column   = 1;
};
```

The last important section of the frame is the code which is placed inside the implementation file of the parser. This section mainly holds the error method for syntactic errors. This method is called by the parser if syntactic errors occur.

C++

```
%{
  #include "Errors.h"
  #include "==>P1Parser.hh"
  #include "==>P1SemEval.h"

  std::istream* yyInStream  = &std::cin;
  std::ostream* yyOutStream = &std::cout;

  extern yy::==>P1Parser::token_type yylex(
           yy::==>P1Parser::semantic_type* yylval,
           yy::==>P1Parser::location_type* yylloc);

  void yy::==>P1Parser::error(
          const yy::==>P1Parser::location_type& l,
          const string& m) {
      GetErrors().SynError(l.begin.line, l.begin.column, m);
  } // yy::==>P1Parser::error
%}
```

Table 4.3: Transformation of EBNF to BNF (based on [Mös87])

EBNF	BNF	
	Variant 1	Variant 2
A = a (b|c).	A → a X X → b | c	
A = a [b].	A → a X X → b | ε	A → a | a X X → b
A = a {b}.	A → a X X → X b | ε	A → a | a X X → b | X b

Table 4.4: Example of LR(1) conflict (based on [Mös87])

EBNF	BNF	
	Variant 1	Variant 2
A = a b | {a} c.	A → a b | X c X → X a | ε	A → a b | c | X c X → a | X a

4.2.2 Grammar Rules

The grammar rules specify the syntactic structure that must be met by the input of the generated compilers. The `NONTERMINAL` section in the parser file defines declarations (prototypes) of the nonterminal symbols. This is necessary, because of allowed cyclic rules. The rules of *Cocol4BoB* are defined in EBNF. The rules in the target file have to be in BNF. Therefore, a transformation from EBNF to BNF is required.

[Mös87] describes a method which replace the EBNF constructs for grouping (), option [] and repetition {} with artificial nonterminal symbols and corresponding rules. The following explanations are based on the referenced work and briefly show how they are applied in *BoB*.

From Table 4.3, it can be seen that there are two variants to perform the transformation. The difference between the two variants is, that variant 1 uses ε rules whereas variant 2 uses additional alternatives. Variant 2 leads to shorter rules and is therefore the preferred. Unfortunately, the epsilon rules also have a downside. In certain situations they lead to LR(1) conflicts.

Table 4.4 shows an example of an LR(1) conflict in variant 1. Reading the symbol `a` results in a shift/reduce conflict. The parser has two options to continue:

Shift – read the next symbol: $A \to ab$ or
Reduce – reduce ε to X: $X \to Xa$

As these conflicts can be resolved by hand, *BoB* uses variant 1 because of the advantage of the shorter rules. The code example below shows a rule in *Cocol4BoB* containing the three constructs described above:

Cocol4BoB

```
SetDecl =
  ident<|str|> '='
  ChrSet { ( '+' | '-' ) ChrSet }
  [ 'IGNORE' ] '.' .
```

The rules of the input file are processed by following *bison* rules:

Bison

```
OptRuleDef:
    /* eps */
  | RULES RuleDefBlock;

RuleDefBlock:
    /* eps */
  | RuleDefBlock RuleDef;

RuleDef:
    IDENT SourceBlock '='
    OptSemDecl OptPreAction
    OptPostAction                { /* SEM: StartRule     */ }
    SynExpr '.'                  { /* SEM: EndRule       */ }
  | error '.';

OptSemDecl:
    /* eps */
  | LOCAL SourceBlock;

OptPreAction:
    /* eps */
  | PRE SourceBlock;

OptPostAction:
    /* eps */
  | POST SourceBlock;

SynExpr:
    SynTerm       { /* SEM: EndProduction/EndCaseExpr     */ }
    SynTermBlock;

SynTerm:          { /* SEM: StartProduction/StartCaseExpr */ }
    SynFact SynFactBlock;

```

```
35 SynFact:
36     IDENT SourceBlock              { /* SEM: AddIdent      */ }
37   | STRING                         { /* SEM: AddString     */ }
38   | SemAction                      { /* SEM: AddSem        */ }
39   | '('                            { /* SEM: StartCompExpr */ }
40      SynExpr ')'                   { /* SEM: EndCompExpr   */ }
41   | '['                            { /* SEM: StartOptExpr  */ }
42      SynExpr ']'                   { /* SEM: EndOptExpr    */ }
43   | '{'                            { /* SEM: StartLoopExpr */ }
44      SynExpr '}'                   { /* SEM: EndLoopExpr   */ }
45   | EPS                            { /* SEM: AddEPS        */ }
46   | RECOVER BY                     { /* SEM: AddRecoverBy  */ };
47
48 SynFactBlock:
49     /* eps */
50   | SynFactBlock SynFact;
51
52 SynTermBlock:
53     /* eps */
54   | SynTermBlock '|'
55     SynTerm           { /* SEM: EndProduction/EndCaseExpr */ };
```

Because the rules can build nested cyclic structures, it is necessary to work with rule and production stacks. Each call to the methods *StartRule* and *StartProduction* leads to a new entry on these stacks. When a production is finished by *EndProduction*, the production is popped from the stack and added to the current rule. When a rule is finished, the method *EndRule* is called and the rule is added to the output. The productions are built by the base components (terminals, nonterminals, etc.) which are added by the *Add...*-methods.

For grouping, option and repetition constructs, the methods *StartCompExpr*, *StartOptExpr* and *StartLoopExpr* create a new artificial nonterminal symbol and add a new rule as shown in Table 4.3. The corresponding *End...*-methods complete the constructs. Figure 4.6 shows the described methods.

Following code examples shows the output of the processed input:

Bison

```
1 SetDecl_1:
2     '+'
3   | '-';
4
5 SetDecl_2:
6     /* eps */
7   | SetDecl_2 SetDecl_1 ChrSet;
8
9 SetDecl_3:
10    /* eps */
11  | IGNORE;
```

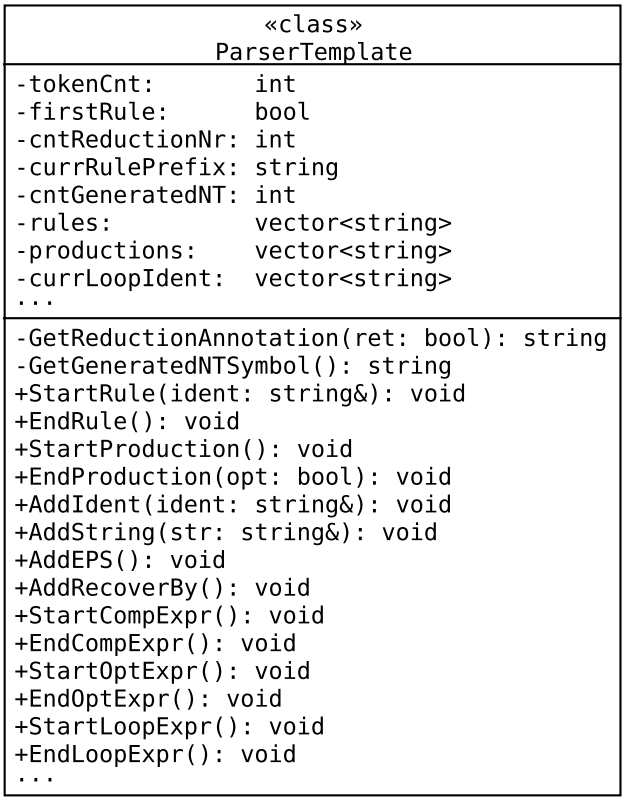


Figure 4.6: *ParserTemplate* – Methods for the treatment of grammar rules

```
12
13 SetDecl:
14     ident '=' ChrSet SetDecl_2 SetDecl_3 '.'
15   | error '.';
```

4.2.3 Error Recovery

One important task of a compiler is error recovery. After an error occurred error recovery puts the parser in an well-defined state from which further syntax errors can be detected. Error recovery has not the attempt to solve syntax errors but to avoid spurious.

Since *BoB* is a *bison* parser, it uses its error recovery mechanism. *Bison* provides special productions which are used if no other production matches. These productions have the form: `error sync-expr`. Within the error state all stack symbols are removed as long the synchronisation expression does not match. The synchronisation points have to be chosen carefully. Normally,

statement separators like `";"` are ideal.

To support error recovery in parsers generated by *BoB*, the compiler description language *Cocol4BoB* is extended by the keyword `RECOVER BY`. It has the same meaning as the `error` keyword in *bison*.

The following code example shows the usage of the error recovery. When an error occurred during recognition of the rule *CommentDecl* all tokens except the synchronisation token defined by the `RECOVER BY` keyword are skipped.

Cocol4BoB

```
1  CommentDecl =
2    'FROM' Delimiter 'TO' Delimiter
3    [ 'NESTED' ] [ 'HANDLER' ident<|str|> ] '.'
4  | RECOVER BY '.' .
```

In the case of an error, it is important to release allocated memory. For this, *bison* provides a `destructor` directive. This directive is used by *BoB*. The generated parser uses another mechanism which is described in section 4.3.1.

4.3 Semantic Evaluator

The generation of the semantic evaluator is the last task of *BoB*. The parts described in the previous sections focus on the lexical and syntactical analysis of the compiler. This section describes the key-components of the semantic evaluator. This includes the collection of the required data during lexical and syntactical analysis and the processing of the data in the semantic evaluator.

Figure 4.7 shows the important parts of the *Cocol4BoB* input file and the resulting sections in the semantic evaluator class. As can be seen, the semantic data is closely linked with the lexical and syntactic analysis. Therefore, the explanations in the following sections include enhancements to the previous sections.

The semantic evaluator consists of three parts, namely

1. the lexical data,
2. the reduction sequence and
3. the semantic methods.

The following sections explain the parts in detail. Figure 4.8 shows the methods of the *SemEvalTemplate* class which produces the parts described in the following sections.

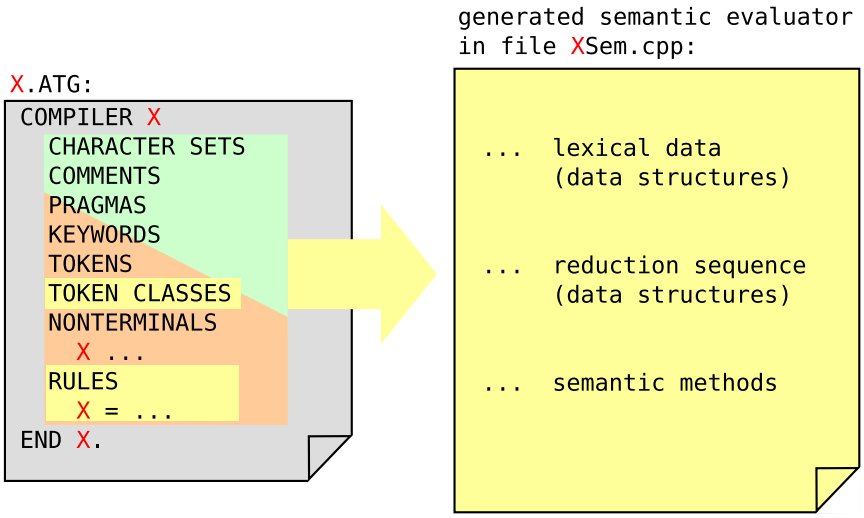


Figure 4.7: General overview of the transformation from *Coco/4Bob* to semantic evaluator

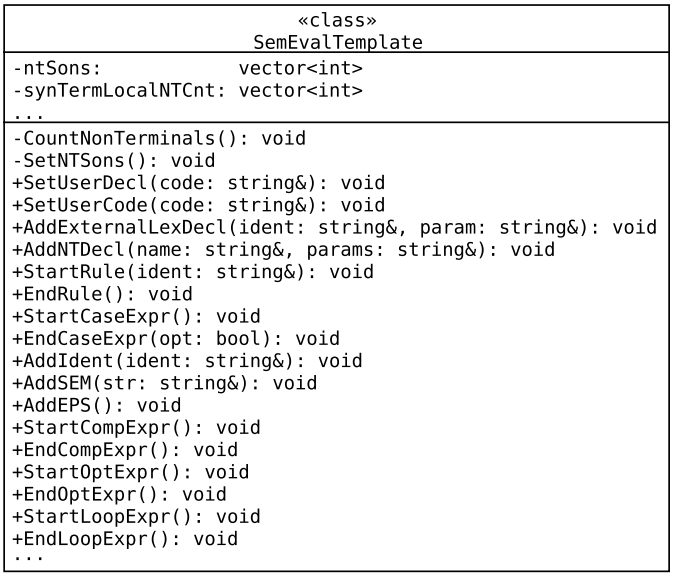


Figure 4.8: *SemEvalTemplate* – Methods for the treatment of the parts which correspond to the processing of lexical data, the creation and transformation of the reduction sequence and the creation of the semantic methods

4.3.1 Processing of Lexical Data

In *Coco/4BoB* there is only one kind of element which can provide lexical data: token classes. Token classes represent terminal symbols with associated

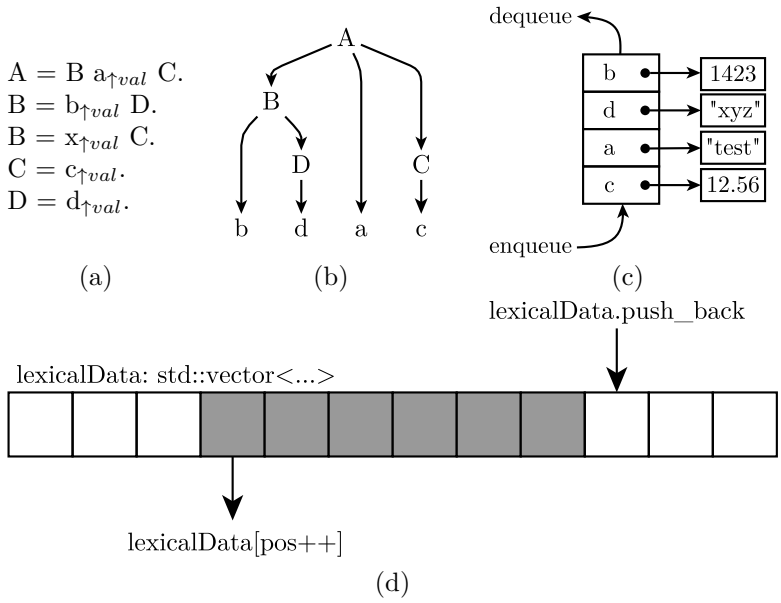


Figure 4.9: Rules (a), syntax tree (b) representation of lexical data (c) and implementation detail – usage of *std::vector* (d) (based on [Mös87])

lexical data.

Figure 4.9 (a) shows a simple grammar which consists of non-terminal symbols and terminal symbols. The terminal symbols have associated attributes which provide the lexical data for the semantic evaluator. The lexical data is needed by the semantic evaluator in the order of occurrence in the input file. The syntax tree for the input `b d a c` shows this by the order of the leaves (see Figure 4.9 (b)). The lexical data is stored in a First-In-First-Out (FIFO) data structure (see Figure 4.9 (c)).

Due to the fact, that the collection and the processing of the lexical data are two time-independent steps, the implementation uses a vector in combination with a position indicator (see Figure 4.9 (d)). The vector is filled by inserting at the rear side. The data is processed by an index which starts at the front side. The used *std::vector* has a better run-time performance as an adequate *std::deque*.

Figure 4.10 shows the relevant parts of the semantic evaluator for processing the lexical data. As described above, the data structure containing the lexical data is provided by the semantic evaluator. Additionally the semantic evaluator provides the method *AddLexicalData* for adding data. For each terminal class, it further provides a method *GET...Attr* for consuming the data by the semantic methods. Following source code shows the implementation of *AddLexicalData* and *GETidentAttr*:

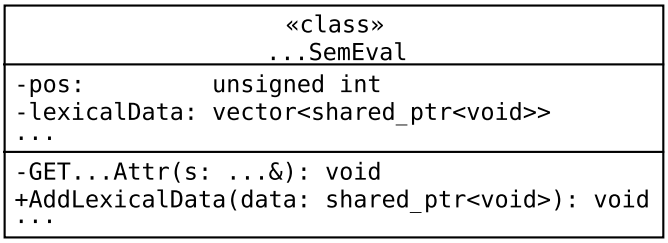


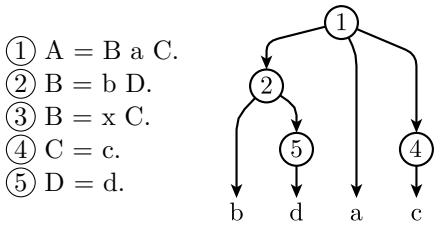
Figure 4.10: Methods/Field for the treatment of lexical data

C++
```
void SemEval::AddLexicalData(shared_ptr<void> data) {
  lexicalData.push_back(data);
} // SemEval::AddLexicalData

void SemEval::GETidentAttr(string& s) {
  s = *static_pointer_cast<string>(lexicalData[pos++]);
} // SemEval::GETidentAttr
```

The lexical data must be collected by the scanner, because this is the component where the data is read in the correct order from the input file. Therefore, the scanner implementation has to be enhanced. In addition to the parts explained in section 4.1.4, an artificial method will be created (*Get...AttrLex*) for each token class. This method contains the various semantic actions which are defined by the user. In the ATG a constant code block is inserted which calls the artificial method and packages the results by calling the method *make_shared*. This packed lexical data is then forwarded to the semantic evaluator by calling the method *AddLexicalData*.

C++
```
void GETidentAttrLex(yy::Parser::location_type& yylloc,
  const char* tokenStr, string& s) {
    s = string(tokenStr);
} // GETidentAttrLex

{letter}({letter}|{digit}|"_")* {
  string s;
  GETidentAttrLex(*yylloc, yytext, s);
  GetSemEval().AddLexicalData(make_shared<string>(s));
  return token::ident;
} // ident
```

The packaging of the lexical data has an additional goal: For correct input files it would not be necessary to use a shared pointer for reference handling. All created data can be destroyed after consumption. So it would be possible to use a `void*` container. However, the input files are not always correct.



(a) (b)

Reduction Sequence					
	Post-Order	5	2	4	1
	Pre-Order	1	2	5	4

NT-Sons	0	2	1	1	0	0
Reduction Number	0	1	2	3	4	5

(c)

Figure 4.11: Rules (a), syntax tree (b) and reduction sequence (c) (based on [Mös87])

Especially in the course of the development of the grammar, there will be errors within the input file. This leads to the abortion of the semantic evaluation. Therefore, the lexical data has to be cleaned up. A `void*` container could not be cleaned up without the information of the contained data. This problem is solved by a shared pointer. A shared pointer saves the information, how to destroy the data, within a delete expression. By deleting the container, each element is destroyed by his associated delete expression.

4.3.2 Construction of the Reduction Sequence

To perform the semantic evaluation in top-down order, the reduction sequence has to be captured during the bottom-up syntax analysis. The basic idea was first presented by [Mös87]. This master thesis uses the idea and adds some modifications to fulfill the requirements of the semantic evaluator.

The reduction sequence is a tree-based structure. The implementation for this master thesis uses a double-ended queue (dequeue or deque). This data structure has two advantages over a tree-based one. First, the memory allocation within a linear data structure is more efficient. There can be allocated bigger chunks of memory at once. Second, a deque needs less memory than a tree.

Figure 4.11 shows the structure of the reduction sequence. An entry in the reduction sequence represents a production in the grammar rules. To accomplish this, each production in the grammar is associated with a number starting with 1. The number 0 has a special meaning: It represents a node with no children and such nodes are usually used to exit loops. The reduction sequence is built by semantic actions added to the productions of the generated parser. Following code example shows the grammar output from section 4.2.2 with the additional semantic actions:

Bison

```
SetDecl_1:
    '+'                     { /* SEM: AddRedNumber(1)     */ }
  | '-'                     { /* SEM: AddRedNumber(2)     */ };

SetDecl_2:
    /* eps */               { /* SEM: AddLoopRedNumber(0)*/ }
  | SetDecl_2 SetDecl_1
    ChrSet                  { /* SEM: AddLoopRedNumber(3)*/ };

SetDecl_3:
    /* eps */               { /* SEM: AddRedNumber(0)     */ }
  | IGNORE                  { /* SEM: AddRedNumber(4)     */ };

SetDecl:
    ident '=' ChrSet        { /* SEM: EnterLoop()         */ }
    SetDecl_2               { /* SEM: LeaveLoop()         */ }
    SetDecl_3 '.'           { /* SEM: AddRedNumber(5)     */ }
  | error '.'               { /* SEM: AddRedNumber(6)     */ };
```

Because each production has a different number of nonterminals, each node has an arbitrary number of sons. Therefore it is necessary, to save the number of sons for each production. This information can be calculated by *BoB* during creation of the compiler.

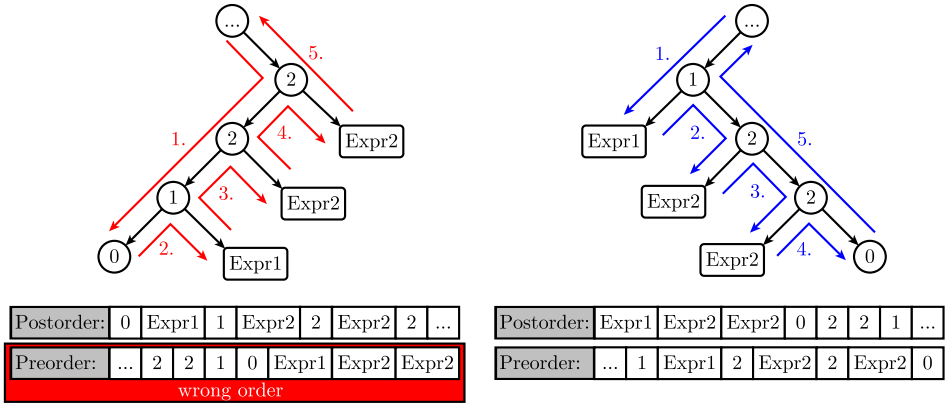
There is one additional problem with the created reduction sequence described so far: Within a bottom-up parser, left recursion is preferred over right recursion, as left recursion needs less space on the stack. Therefore, *BoB* uses left recursion for implementing loops (repetition in EBNF $\{\ldots\}$ – see Figure 4.12 (a)). The reduction sequence created by left recursion (see Figure 4.12 (b)) is different from that created by right recursion (see Figure 4.12 (c)). For the semantic evaluator it is important, that the loop reduction number and the reduction numbers which are contained in the loop are evaluated successively, because otherwise the processing order of the reduction numbers is disturbed. The correct order is guaranteed only by the reduction sequence as created by right recursion (see Figure 4.12 (d)). Therefore, a transformation from left to right recursion within loops is required.

As seen in Figure 4.13 this transformation is divided into three steps:

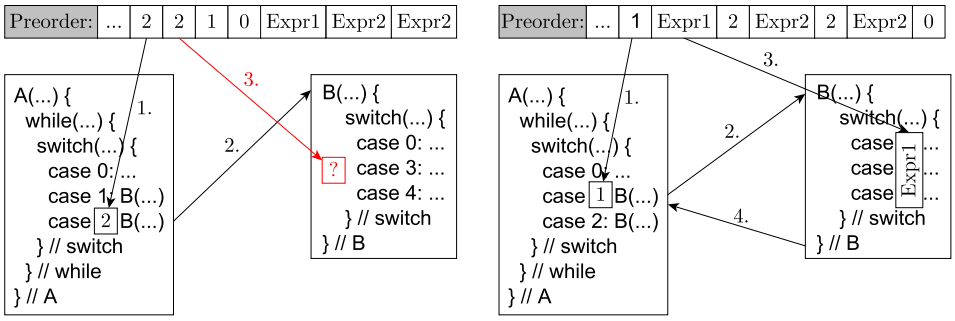
1. In the first step, all subsequent loop calls are added to an temporary buffer. All other nodes within the loop are added directly to the resulting reduction sequence. When the loop is finished, the next two steps are performed.
2. In step 2, the first entry in the buffer is added to the reduction sequence.
3. And finally, in step 3 the last elements in the buffer are added in reverse order to the reduction sequence.

EBNF	BNF (left recursion)	BNF (right recursion)
Loop = { Expr1 | Expr2 }.	Loop: Loop_1; Loop_1: /* eps */ ⓪ | Loop_1 Expr1 ① | Loop_1 Expr2; ②	Loop: Loop_1; Loop_1: /* eps */ ⓪ | Expr1 Loop_1 ① | Expr2 Loop_1; ②

(a)



(b) (c)



(d)

Figure 4.12: Transformation of repetition from EBNF to BNF with left and right recursion (a), reduction sequence of loop created by left recursion (b) and by right recursion (c), processing of the reduction sequence created by left and right recursion – reduction sequence by left recursion is not processable by top-down semantic methods (d)

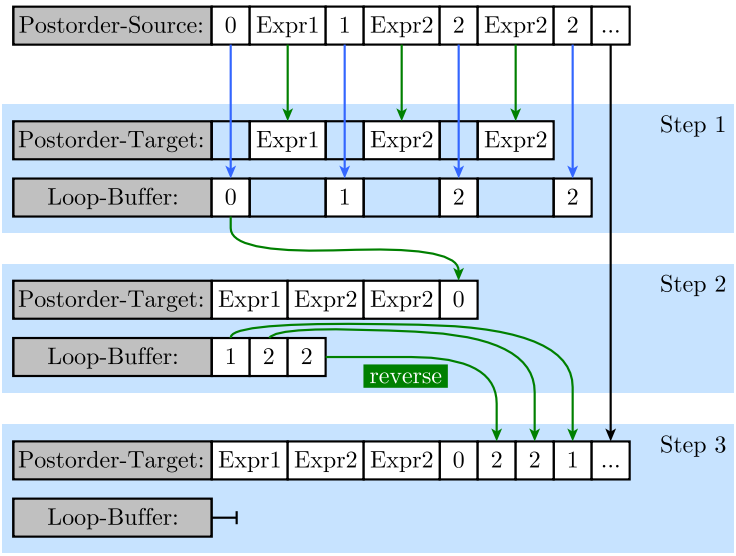


Figure 4.13: Transformation of the reduction sequence – from left to right recursion

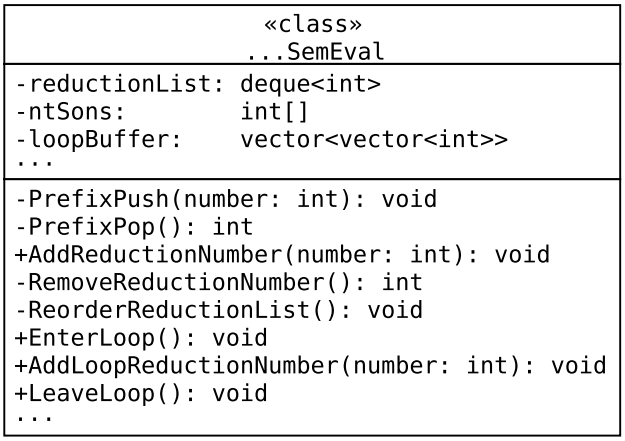


Figure 4.14: Methods/Data components for the creation/transformation of the reduction sequence

Figure 4.14 shows the methods used for the creation of the reduction sequence explained in this section. The first step is performed by the method *EnterLoop*. Steps 2 and 3 are performed by method *LeaveLoop*. The recursive calls in the loop are added by the method *AddLoopReductionNumber*. All other productions are added by the method *AddReductionNumber*. An example is shown in the code example on page 50.

4.3.3 Transformation of the Reduction Sequence

After the syntactic analysis, the reduction sequence is available in post-order. This represents the order generated by bottom-up parsing. For the semantic evaluation, the sequence has to be transformed to pre-order which is the necessary order for top-down processing. [Mös87] presented the following recursive solution:

C++
```
void ...SemEval::ReorderReductionList(int prodNumber) {
  for (int i = 0; i < ntsons[prodNumber]; i++) {
    ReorderReductionList(RemoveReductionNumber());
  } // for
  PrefixPush(prodNumber);
} // ...SemEval::ReorderReductionList
```

The method *ReorderReductionListRec* is called for the root of the reduction sequence. Afterwards, for each son a recursive call to *ReorderReductionListRec* takes place. The son is obtained by a call to *RemoveReductionNumber*, which delivers the next reduction number from the reduction sequence. The reduction sequence is read in the reverse order, beginning at the end. The number of sons of the current node is determined by the *ntsons* array. Finally, the reordered reduction sequence is built by adding the current reduction number to the front of the new reduction sequence by the method *PrefixPush*. All methods are shown in Figure 4.14.

For *BoB* the recursive algorithm is replaced by an iterative one. The recursive version has problems with large reduction trees due to the limited stack memory. The iterative algorithm replaces the recursive calls with stack-entries on the heap. Following code example shows the iterative version:

C++
```
void ...SemEval::ReorderReductionList() {
  int i, prodNr;
  vector<tuple<int, int>> stack;

  stack.push_back(make_tuple(0, RemoveReductionNumber()));
  do {
    i =      get<0>(stack.back());
    prodNr = get<1>(stack.back());
    stack.pop_back();
    while (i < ntsons[prodNr]) {
      stack.push_back(make_tuple(++i, prodNr));
      prodNr = RemoveReductionNumber();
      i = 0;
    } // while
    PrefixPush(prodNr);
  } while (!stack.empty());
```

```
17 } // ...SemEval::ReorderReductionList
```

4.3.4 Semantic Methods

For each grammar rule, *BoB* creates a semantic method which executes the semantic evaluation. The first grammar rule defines the main method. This method is called after the lexical and syntactical processing of the input is finished. The semantic methods are similar to recognition functions created by a compiler using the method of recursive descent [Aho+06]. The difference is that no lexical and syntactical actions are performed, as these have been executed already during lexical and syntactical analysis. The following source code example shows an example grammar rule with the relevant parts for the semantic method:

Cocol4BoB

```
1  A<|[1]|> =
2    LOCAL<|[2]|>
3    PRE  <|[3]|>
4    POST <|[4]|>
5    a<|[6]|> SEM<|[5]|>
6    B<|[7]|> SEM<|[5]|>   (1)
7  | ( b<|[6]|> (5) )      (2)
8  | [ C<|[7]|> (6) ]      (3)
9  | { c<|[6]|> (7)
10   | d<|[6]|> (8) }      (4)
11 .
```

The circled numbers (1)–(8) represent the reduction numbers of the productions. And the boxed numbers [1]–[7] represent the relevant data which is extracted from the grammar rules for creation of the semantic methods. The following list explains the used data:

- [1] Formal parameter list of semantic method.
- [2] Local semantic declarations.
- [3] Semantic action interleaved at the start of the semantic method.
- [4] Semantic action interleaved at the end of the semantic method.
- [5] Semantic action – before, after or between symbols.
- [6] Actual parameter list for *Get...Attr* method, to get the lexical data.
- [7] Actual parameter list for calls of semantic methods.

The generated semantic method basically contains a `switch` statement to select the different productions. The control flow is defined by the reduction sequence. Within the switch condition, a call to the method *NextRedNr*

delivers the next reduction number and selects the next alternative. Within an alternative, following components occur:

- For each **terminal class**, a call to the Method *Get...Attr* is inserted. As described in section 4.3.1, this method delivers the captured lexical data.
- For each **non terminal symbol**, a call to the associated semantic method is inserted.
- The **semantic actions** are taken directly from the grammar.
- A **grouping** construct (EBNF) is realized by a `switch` statement.
- An **option** construct (EBNF) is also implemented with a `switch` statement and has an additional ε alternative.
- The **repetition** construct (EBNF) is realized like the option construct. Additionally, the construct is surrounded by a `while` statement, which enables the recursive calls. The ε alternative is used as exit point from the loop.

As can be seen for the artificial rules (grouping, option and repetition), no separate semantic methods are created. These constructs are realized as embedded statements which act like separate methods. This has the advantage, that there are no scope problems with semantic variables.

The source code below shows the semantic method generated from the grammar rule above:

C++

```
1  void A(1) {
2    2 3
3    bool done = false;
4    switch (NextRedNr()) {
5      case 1:
6        GETaAttr(6); /*SEM*/ 5 /*SEM*/
7        B(7);        /*SEM*/ 5 /*SEM*/
8        break;
9      case 2:
10       switch (NextRedNr()) {
11         case 5:
12           GETbAttr(6);
13           break;
14       } // switch
15       break;
16     case 3:
17       switch (NextRedNr()) {
18         case 0:
19           break;
20         case 6:
```

```
21            C(7);
22            break;
23        } // switch
24        break;
25      case 4:
26        while (!done) {
27          switch (NextRedNr()) {
28            case 0:
29              done = true;
30              break;
31            case 7:
32              GETcAttr(6);
33              break;
34            case 8:
35              GETdAttr(6);
36              break;
37          } // switch
38        } // while
39        break;
40    } // switch
41    4
42  } // A
```

4.4 Summary

This chapter gave an insight into the implementation of *BoB*. *BoB* is composed of three main parts. These were discussed in the previous sections. First, the transformation from *Cocol4BoB* to *flex* input was explained. Subsequently, the transformation from *Cocol4BoB* to *bison* input was described. Finally, the components of the semantic evaluator were described.

The next chapter describes the tests carried out and discusses the results. Furthermore, the results of this work are evaluated and discussed.

Chapter 5

Tests, Evaluation and Discussion

This chapter gives an overview of the tests performed. Moreover, the result of this work is evaluated and discussed. Firstly, the *BoB* workflow is explained. Secondly, the test data and process are presented. After that, the bootstrapping used for testing is explained. The next sections present the results of the performance analysis. The next sections deal with the evaluation and discussion: *BoB* is compared with existing compilers in terms of usability, capability and performance. Finally, shortcomings, restrictions and open issues are discussed.

5.1 *BoB* Workflow

For creating a compiler with *BoB*, several steps are required. Figure 5.1 shows the workflow for *BoB*. First, the user has to create a *Cocol4BoB* file (ATG) which holds the compiler description. From this input file *BoB* creates four output files. These are the scanner description for *flex*, the parser description for *bison* and the .h/.cpp files of the semantic evaluator. Moreover, the user can intervene in the translation process by modifying the frame files provided by *BoB*.

The next step is to create the scanner and parser files from the generated description files. For this task, *flex* and *bison* are used. These two transformations can run independently. Now, all generated files are available. However, there is still missing a main file. This can be derived from the *Main.frm* delivered by *BoB*, which contains a standard implementation. Another option is to provide the main file by the user himself.

When the whole source files of the new compiler are generated, these files have to be compiled by any common C++ compiler like *gcc* or the *MS C/C++ Compiler*. An important note is that *flex*/*bison* basically generate *gcc* compatible code. Therefore, a special version called *Win flex-bison* is used. This version supports the command line option `--wincompat` which changes the unix include `<unistd.h>` to the analogous windows include `<io.h>`.

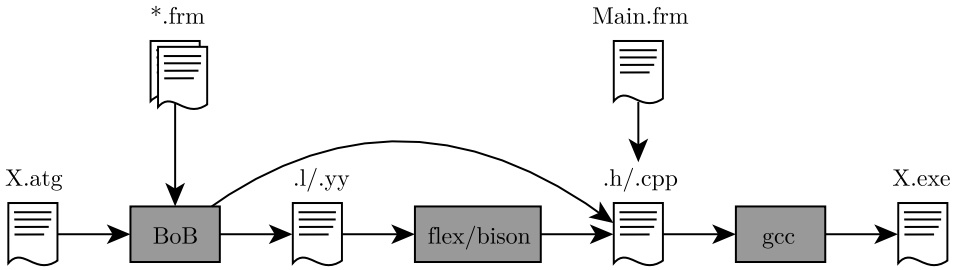


Figure 5.1: Workflow of *BoB*

Table 5.1: ATGs for verification and validation

ATG	Size	Compiler Generator	Purpose
BoB	Large	*BoB*	Verification
Calc	Small	*BoB*, *Coco-2*, *flex/bison*	Validation
MiniC	Medium	*BoB*, *Coco-2*, *flex/bison*	
MiniCpp	Large	*BoB*, *Coco-2*, *flex/bison*	

Because the workflow requires multiple steps, it is important to provide a convenient automation of the process. For this, there are several solutions:

- The user can use a shell/batch script or similar.
- However, during development of *BoB* the cross-platform, open-source build system *CMake* is used. *CMake* is configured by text files and allows to define the complete compilation process. *CMake* supports many third party tools out of the box, like *flex* and *bison*. Moreover it is easy to extend *CMake* with custom tools like *Coco-2* (for automated comparison tests). From the configuration files *CMake* generates platform dependant makefiles or workspaces. [MH10] provides a detailed description of *CMake*.

5.2 Test Data and Test Process

BoB was tested by using example attributed grammars (ATGs) for generating test compilers. The selected ATGs differ in complexity, size and purpose. Table 5.1 shows the used ATGs: *BoB.atg* is used to perform bootstrapping (see section 5.3), the other ATGs are real world examples to compare *BoB* with other compiler generators.

All tests are carried out automatically by *CMake*. *CMake* provides the testing tool *CTest*. This tool provides the `add_test` command which can be used

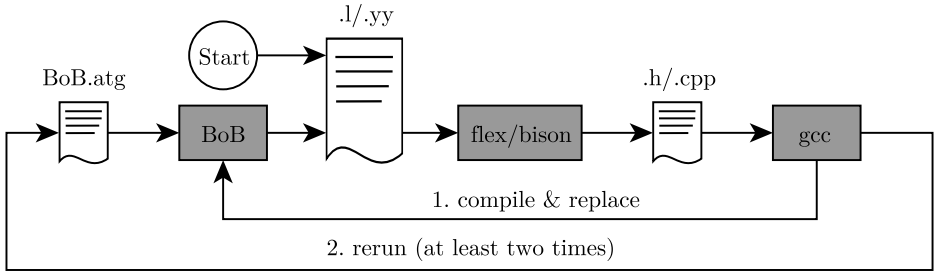


Figure 5.2: Workflow for bootstrapping of *BoB*

to execute tests on the generated compilers. *BoB* is tested in two steps: The first step generates the compilers from the ATGs as described above. In the second step the generated compilers are tested by various test inputs. Therefore, changes to *BoB* requires the following execution sequence:

1. Compile *BoB*.
2. Test *BoB* – generates files of compilers (*BoB*, *Calc*, *MiniC* and *MiniCpp*).
3. Compile generated compilers.
4. Test generated compilers.

The testruns of the generated compilers tell only that the input files have been processed without errors. The functionality check is done by (manually) checking the output of these tests.

5.3 Bootstrapping

BoB is verified by bootstrapping. Figure 5.2 shows the bootstrapping workflow:

1. First, the preprocessor *BoB* has to be generated. For this task the provided input files for *flex* and *bison* are used.
2. After the initial step, *BoB* is used to generate itself by the provided ATG (see Apendix A).

BoB generated by the initial step and *BoB* generated by the subsequent iterations differ in the internals. *BoB* of the initial step is a clean *flex/bison* generated compiler. *BoB* from the subsequent iterations is a *BoB-flex-bison* generated compiler which supports bottom-up syntax analysis and top-down semantic evaluation. The verification is done by comparing the outputs of the two different preprocessors for equality.

5.4 Run-time Analysis

An important quality criterion is the run-time of the compiler generators and especially of the generated compilers. The following sections compare the new tool-chain *BoB-flex-bison* with *Coco-2* and *flex/bison*. Two cases are of interest: on the one hand the run-time of the compiler generators, and on the other hand the run-time of the generated compilers.

5.4.1 Run-time of *BoB*

For averaged sized grammars, the *BoB-flex-bison* toolchain results in a mean run-time of 200 ms (Test system: Intel Core 2 Duo E6750, 2 x 2.66 GHz, 4 GB RAM, Windows 8 64-Bit). To measure the run-time overhead by *BoB*, the *BoB-flex-bison* toolchain was compared with the similar *flex/bison* toolchain. This measurement shows, that the additional preprocessing through *BoB* leads to an run-time overhead of about 20 %.

5.4.2 Run-time of Compilers generated by *BoB*

The compilers *Calc*, *MiniC* and *MiniCpp* are generated with each of the compiler generators *BoB-flex-bison*, *Coco-2* and *flex/bison*. For the comparison of the run-times, each of this generated compilers is called with test files, which contain thousands of terminal symbols. The comparison of *Calc* and *MiniC* shows that *Coco-2* and *flex/bison* generated compiler(s) are almost equally fast and *BoB* generated compiler(s) require five times of the run-time as *Coco-2* and *flex/bison* generated compiler(s). *Calc* and *MiniC* are simple test-cases with less complexity than *MiniCpp*. Therefore, the test-case with *MiniCpp* is more meaningful.

The diagram in Figure 5.3 shows the run-time comparisons for the compiler *MiniCpp* compiled with *gcc*. As can be seen, the compiler generated by *flex/bison* requires only a tenth of the run-time consumed by *BoB* generated compilers. Moreover it is noteworthy that compilers generated by *BoB* need 30 % more run-time than those generated by *Coco-2*.

The diagram in Figure 5.4 shows the same tests with a different compiler – the *MS C++ Compiler*. It is noteworthy, that the run-time of *BoB* compiled with the *MS C++ Compiler* is significantly higher. This difference mainly comes from the different block size of the *deque* implementation of the *gcc* and *MS C++* shared library. *Gcc* uses a block size of `512 * size_of_value_type` and *MS C++* uses a maximum of `16 * size_of_value_type`. This small block size is good for few elements, but leads to a slow down with many elements.

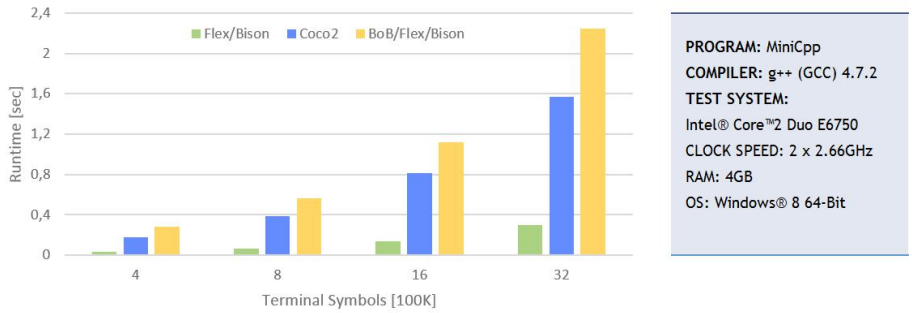


Figure 5.3: Comparison of *MiniCpp* generated by *BoB-flex-bison*, *Coco-2* and *flex/bison* – compiled with *gcc*

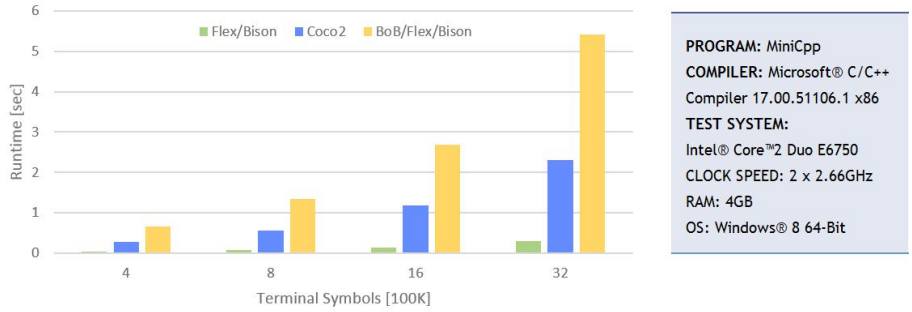


Figure 5.4: Comparison of *MiniCpp* generated by *BoB-flex-bison*, *Coco-2* and *flex/bison* – compiled with *MS C++ Compiler*

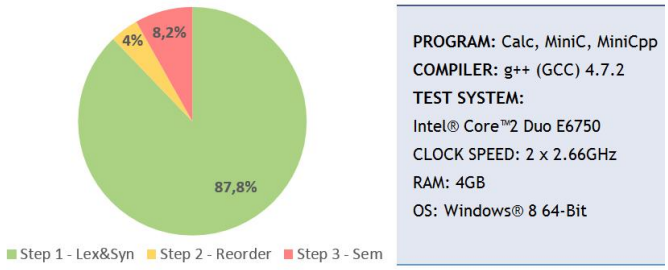


Figure 5.5: Run-time distribution of the main parts of the generated compilers – average over *Calc*, *MiniC* and *MiniCpp*

Figure 5.5 shows the average distribution of the run-times of the main phases of the generated compilers. The lexical and syntactical analysis takes the main part of the time. This is attributable to the computation of the lexical data and the reduction sequence. The reorder and the semantic evaluation steps consume only a fraction of the lexical and syntactical analysis steps. It is noteworthy that the run-time of the semantic evaluation step

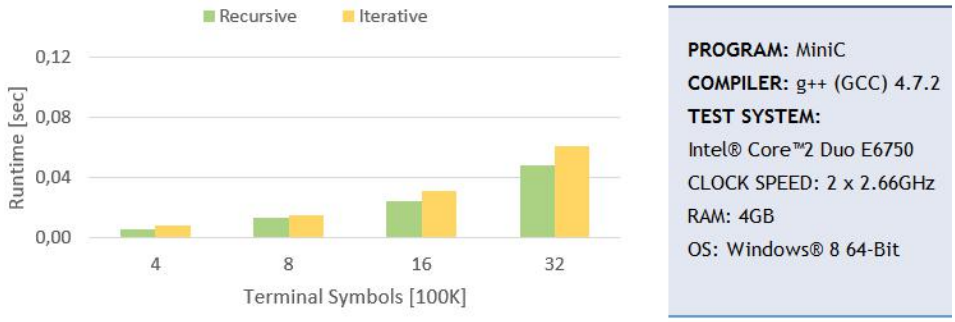


Figure 5.6: Run-time comparison of iterative and recursive *Reorder*

depends highly on the used semantic actions. Nevertheless, this measurement indicates that the advantage of termination due to an error at the lexical and syntactical analysis leads to a time saving of about 12 %. Generally, run-time improvements should be made, especially in the lexical and syntactical analysis steps.

Figure 5.6 shows the run-time analysis of the reorder step. As described in section 4.3.3 there are two solutions for reordering the reduction sequence: The iterative version is preferable because it does not stress the run-time stack (but uses a data structure). However, the run-time analysis shows that the recursive version is about 8 % faster. In conclusion, it can be stated, that the iterative version is more secure, but for common problem sizes the recursive version is preferable.

5.5 Analysis of Memory Usage

Coco-2 and *flex/bison* generated compilers have a constant memory consumption of about 350 KB (compiler: *g++* (GCC) 4.7.2)). The reason for this is, that these compilers process the input on the fly without intermediate data structures.

Figure 5.7 shows the maximal memory consumption of *MiniCpp* generated by *BoB*. It is obvious, that there is a linear relationship between file size and memory consumption. The memory requirement can be explained by the creation of the intermediate representation between the lexical/syntactical analysis and the semantic evaluation.

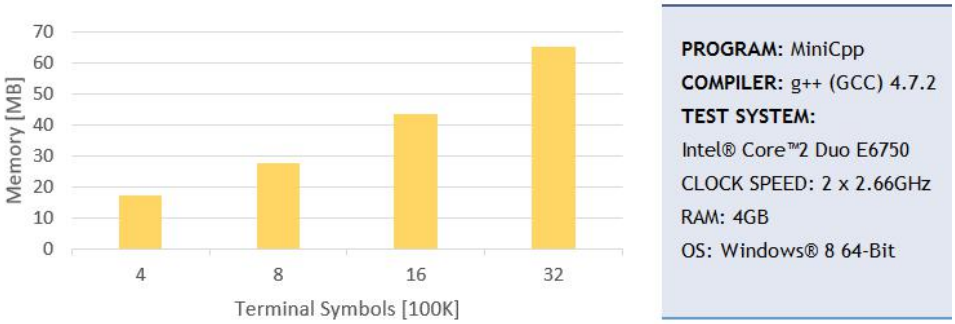


Figure 5.7: Maximal memory consumption of *MiniCpp* generated by *BoB*

Table 5.2: Comparison of existing compiler generators

Compiler Generator	Target Language(s)	Attributed Grammar	
		Power	Attribution
Coco-2	Pascal, Modula-2, Ada, C, C++, Java, C#	LL(1)	L-attr.
flex/bison	C, C++, Java	LALR(1)	S/(LR)-attr.
Smart	Modula-2	LALR(1)	L-attr.

5.6 Evaluation of Results

The main objective of this master thesis was to combine the advantages of bottom-up syntax analysis with the advantages of top-down semantic evaluation. Because of this objective, existing compiler generators were first analyzed. Table 5.2 shows the important characteristics of these compiler generators. Based on this preliminary work, the decision was taken to develop a preprocessor for *flex* and *bison*.

As input for the preprocessor *BoB* the compiler description language *Cocol4BoB* was specified. *BoB* translates the input from *Cocol4BoB* into input files for *flex* and *bison*. *Flex* and *bison* are used to generate a lexical analyzer and a bottom-up syntax analyzer. Moreover, *BoB* creates a top-down semantic evaluator.

Therefore, the new compiler toolchain *BoB-flex-bison* combines the advantages of the simple input language of *Coco-2*, the powerful bottom-up syntax analysis of *flex/bison* and the semantic analysis concepts of *Smart*. With this implementation, the goals set and the derived requirements (see section 1.3) were satisfied. The following sections discuss the chosen implementation.

5.7 Comparison of *BoB* with *Coco-2* and *flex*/*bison*

To assess the quality of the implemented preprocessor *BoB*, a comparison with existing compiler generators is necessary. Hence, the pros and cons of the (generated) compilers in terms of usability, capability and performance are compared.

5.7.1 Usability

The usability depends essentially on the selected compiler description language, since this is the user interface of a compiler generator. *Coco-2* uses a language called *Cocol-2*. *Cocol-2* was optimized in terms of simplicity and usability. This is achieved primarily through the use of a single input file and the use of EBNF for the description of the scanner and the parser.

On the other hand, *flex* and *bison* use separate files for the scanner and parser description. The scanner is described by regular expressions. The parser is described by BNF. This leads to the disadvantage that duplicate declarations are necessary, which can lead to inconsistency. Moreover, BNF is less expressive than EBNF used by *Cocol-2*.

BoB uses an adapted version of *Cocol-2* called *Cocol4BoB*. This version includes adjustments to the target language C++ and to the compiler generators *flex* and *bison*. Thus, it can be stated that *Cocol4BoB* was built on an already established language and provides all its advantages.

Another point concerning the usability is the length of the toolchain. Since *BoB* is an additional component, it increases the complexity. This problem can be solved with automation by appropriate batch/shell scripts or by using a build tool like *CMake*.

5.7.2 Capability

The capability of a compiler generator is defined by the power of the ATGs and the supported attributes in semantic actions.

Coco-2 supports LL(1) grammars only. LL(1) grammars are usually sufficient, but there are languages that can not be described with this class of grammars. *Coco-2* supports L-attributed grammars. Thus, inherited and synthesized attributes are allowed within semantic actions.

In contrast to *Coco-2*, *flex* and *bison* handle LALR(1) grammars. LALR(1) grammars are a superset of LL(1) grammars and can therefore describe a

larger class of languages. On the other hand, *flex* and *bison* only support the S-attribution or a kind of LR-attribution with global variables.

The presented compiler generators have their strengths and weaknesses regarding chain power. *BoB* has set out to combine the strengths of them. Therefore, *BoB* supports L-attributed LALR(1) grammars. In this respect *BoB* is superior to other compiler generators. Thus, *BoB* lowers the initial hurdles for aspiring compiler developers.

5.7.3 Performance

The advantages of *BoB* described in the previous sections are not for free. The main drawback of *BoB* is the minor performance during creation of the compiler. Moreover, the performance of the generated compiler is significantly worse than with *Coco-2* and *flex/bison*.

As described in section 5.4, the generation with *BoB* takes up to 20 % more time. Moreover, compilers generated by *flex/bison* are about 10 times faster. Compilers generated by *Coco-2* are up to 30 % faster.

Another aspect is the memory usage. Whereas *Coco-2* and *flex/bison* have a constant memory complexity, *BoB* has a linear complexity.

The developers have to decide which characteristics are more important for their applications. There are applications where the benefits of *BoB* exceed the drawbacks in time and memory usage.

5.8 Shortcomings and Restrictions

The targets defined at the beginning of this master thesis could be largely met. However, certain restrictions had to be made to avoid going beyond the scope of this work.

- *BoB* does not support the `ANY` symbol introduced by *Smart* and *Coco-2*. Within *Smart* the `ANY` symbol is defined as follows [Mös87]: The complement symbol `ANY` means any terminal symbol, which is not the terminal beginning of the alternative chain to that the `ANY` symbol belongs. Thus, `ANY` is a wildcard for elements of an entire terminal symbol set.
- *BoB* supports only the error recovery mechanism *special catch symbols*. This method leads to ignoring a large amount of symbols without checking their syntactical correctness. [SSS90] shows advanced concepts of error recovery during bottom-up parsing. Nevertheless, the integration into *bison* is not trivial.

- *BoB* uses a C++ compatible scanner which is written in C. This contradicts the object-oriented design of *BoB* and is therefore suboptimal. *Flex* can create a C++ scanner, but this feature is in an experimental state.
- Scanners and parsers generated by *BoB* are not reentrant. This leads to the restriction that the generated scanners and parsers can not be instantiated multiple times. Because multiple instances would interfere each other.

Despite these limitations, *BoB* is a fully functional preprocessor to generate a compiler which supports bottom-up syntax analysis and top-down semantic evaluation. Moreover, all of these restrictions can be overcome by further development.

5.9 Open Issues

There are some open issues which are not treated by this master thesis. These relate mainly to the optimization of the existing implementation. As outlined in section 5.4 the performance of *BoB* is worse than that of comparable compiler generators. There are three points to improve performance:

1. The rules generated by *BoB* contain duplicates. This leads to bigger grammars and therefore larger automata. The removal of these duplicate rules may lead to a run-time improvement. Even more tweaks to the grammar are conceivable. For example, rules with just a single production could be optimized away.
2. If syntactic errors occur, *BoB* is terminated after the lexical and syntactical analysis. This first steps require most of the run-time. Therefore, only little time is saved. This could be improved by relocating the time-consuming processing of the lexical data into the semantic evaluation step. But this solution raises problems concerning the treatment of lexical data that has to be processed by the scanner.
3. The data structure of the reduction sequence is optimized in terms of memory consumption. Due to the storage in a sequential container, a transformation from post- to pre-order is needed. By storing the reduction sequence in a tree structure, this transformation could be replaced by a tree traversal. It is difficult to predict whether the overhead in the generation of the tree or the reorder step consumes more run-time.

It should be noted, that despite these improvements the performance of a pure bottom-up or top-down parser will not be achieved.

5.10 Summary

This chapter in the first part gave an overview how the functionality of the implemented preprocessor *BoB* can be tested. At the beginning, the *BoB*-workflow was explained. After that, the test data and testing procedure was described. Moreover, bootstrapping was explained.

The next part dealt with the results of the performance analysis. This section covered a detailed comparison of the consumed run-time and memory usage.

The third part of this chapter performed an evaluation of the implementation. First, the results were discussed again shortly. Next, the implementation was compared with other compiler generators. Finally, the shortcomings, restrictions and open issues of the chosen solution approach were presented.

Chapter 6
Summary, Outlook and Experience

This chapter concludes the master thesis. First, the main aspects of the work are summarized. In addition, specific applications and the significance of the results are discussed. The next section gives an outlook of possible extensions to the work. The chapter ends with a description of the personal experiences of the author.

6.1 Summary

The aim of this master thesis was to combine the bottom-up syntax analysis with the top-down semantic evaluation. This approach has the advantage that L-attributed LALR(1) grammars can be processed.

Starting from the basic objectives, the fundamentals were explored. Based on this knowledge, the development of the preprocessor *BoB* was started. During the development of the preprocessor, the specification of the compiler description language *Cocol4BoB* was made. *Cocol4BoB* is characterized by a particularly simple and powerful syntax.

The development was divided into two main parts: The first part concerned mainly the transformation of *Cocol4BoB* constructs into *flex/bison* equivalent constructs. This part was very time-consuming, because solutions had to be invented which perform the transformations in an automatic way.

The second part concerned the semantic evaluator. The semantic evaluator provides the functionality to collect the semantic data during bottom-up lexical and syntactical analysis. With this data, the semantic evaluator can process the semantic actions in a top-down order.

The goals were largely achieved. The implemented preprocessor is fully functional. Only the performance should be improved.

6.2 Fields of Application

The developed preprocessor *BoB* can be used everywhere, where other compiler generators can be used. Since he only can generate compilers in C++ it is restricted to this target language. *BoB* supports the powerful L-attributed LALR(1) grammars. The compiler developer has not to care for LL(1) conflicts and can use the intuitive inherited and synthesized attributes. Therefore, *BoB* is the ideal compiler generator toolchain for beginners.

Moreover, the generated *flex* and *bison* files can be used as basis for custom compilers. The generated C-scanner is optimized to interact with the C++ parser. This is not easy to implement, and can thus serve as a basis for own developments.

6.3 Outlook

The original objectives of the master thesis have been achieved. Nevertheless, there are some points which could not be addressed in the context of this master thesis. In addition to the shortcomings, restrictions and open issues outlined in the sections 5.8 and 5.9 there are following points for further developments:

- A major disadvantage of the current implementation, is the dependence on the target language C++. This dependence is due to the usage of *flex* and *bison*. To reduce this dependence, there could be used two strategies: The first one is to generate input files for another bottom-up parser which supports another target language. The second approach is to develop a custom bottom-up parser which has a changeable back-end for different target languages.
- *Cocol-2* and *Cocol4BoB* grammars are largely identical. To reuse existing compiler descriptions in *Cocol-2*, a transformation tool from *Cocol-2* to *Cocol4BoB* and vice versa would avoid the error prone transformation by hand.
- A graphical user interface (GUI) could decrease the entry barrier to the command-line tools. This GUI could support following features:
  - Syntax Highlighting for *Cocol-2* and *Cocol4BoB*.
  - Running the *BoB-flex-bison* toolchain.
  - Compile the generated files with different compilers, like *gcc* and *MS C/C++Compiler*.
  - Presentation of detailed error messages with the possibility to open the error position within the source document.

This extension could be realized by using the *Eclipse Rich Client Platform* (RCP). [DFK05] gives an good insight into the RCP. For the topic of syntax highlighting [Dic11] gives a sample implementation.

The points listed above provide approaches to further developments and do not limit the current functionality of *BoB*. They would extend the application field of *BoB*. With *BoB*, a powerful approach is available which can be taken up by other compiler generators.

6.4 Personal Experience

During the study program (Software Engineering Master), I learned the basics of the used technologies within the lecture *Formale Sprachen, Compiler- und Werkzeugbau*. However, more in-depth knowledge about the functionality of *flex* and *bison* were required for this master thesis. Especially at the begin of this master thesis the knowledge search required very often.

It was essential, that the parts of *BoB* could be separated into independent parts. At the beginning, I created the input files for *flex* and *bison* which could process *Cocol-2* grammar files. These files were used as target for the files to be generated by *BoB*. Then, the transformations of *BoB* were gradually developed and compared with the targets.

After completion of the parts of *BoB* which generates the scanner and parser, the same process was used for the implementation of the semantic evaluator. Due to the novelty of the approach, several solutions have been tried and rejected. Finally, a solution was found, which solves the problem adequately.

I found it particularly interesting to engage myself with the programming language C++. Especially, the new concepts of the C++11 standard often allowed more compact and readable source code. Another highlight was the cross platform build tool *CMake*. It helped me to generate platform dependant make files and workspaces for *Eclipse* and *Visual Studio*. Moreover, with *CMake* the system tests were automated.

Appendix A

Cocol4BoB Grammar of *BoB*

The following example is the compiler description of *BoB*. This *Cocol4BoB* example can be used for bootstrapping. Due to space limitations the lexical methods are shortened to their declarations. Moreover the calls in the semantic actions are replaced by comments.

Cocol4BoB

```
COMPILER BoB

LEX<|
  /* pragma handling */
  #include <sstream>
  using std::ostringstream;

  #include <unordered_set>
  using std::unordered_set;

  unordered_set<string> defined;
  unordered_set<string> undefined;
  int endOrElseRequired = 0;
  int endsRequired = 0;

  static string HandleSourceCode(...) {...}
  static string ReadIdentifier(...) {...}
  static void IgnoreIfCond(...) {...}
  static void IgnoreElseCond(...) {...}

  static void HandleDEF(...) {...}
  static void HandleUNDEF(...) {...}
  static void HandleIF(...) {...}
  static void HandleELSE(...) {...}
  static void HandleEND(...) {...}

  static bool SLHandler(...) {...}
  static bool MLHandler(...) {...}
|>
```

```
30
31 SEMDECL<|
32   #include <string>
33   using std::string;
34 |>
35
36 SEM<|
37   #include "Errors.h"
38   #include "ScannerTemplate.h"
39   #include "ParserTemplate.h"
40   #include "SemEvalTemplate.h"
41
42   static string str = "";
43   static int num = 0;
44 |>
45
46 CHARACTER SETS
47   letter   = 'A'..'Z' + 'a'..'z'.
48   digit    = '0'..'9'.
49   control  = CHR(0) .. CHR(31) + CHR(127).
50   noQuote1 = ANY - "'" - control. (*valid characters in '...'*)
51   noQuote2 = ANY - '"' - control. (*valid characters in "..."*)
52   ignore   = EOL + CHR(9) IGNORE.
53
54 COMMENTS
55   FROM '--' TO EOL          HANDLER SLHandler.   --Ada comments
56   FROM '//' TO EOL.                               //C++ comments
57   FROM '/*' TO '*/'         HANDLER MLHandler.   /*C comments*/
58   FROM '(*' TO '*)' NESTED.                      (*Modula-2 comments*)
59
60 KEYWORDS
61   'ALIAS'.      'ANY'.        'BY'.             'CASE'.
62   'CHARACTER'.  'CHR'.        'CLASSES'.        'COMMENTS'.
63   'COMPILER'.   'END'.        'EOL'.            'EPS'.
64   'FOLLOWED'.   'FROM'.       'HANDLER'.        'IF'.
65   'IGNORE'.     'KEYWORDS'.   'LEX'.            'LEXLOCAL'.
66   'LOCAL'.      'NESTED'.     'NONTERMINALS'.   'PRAGMAS'.
67   'PRE'.        'POST'.       'RECOVER'.        'RULES'.
68   'SEM'.        'SEMDECL'.    'SETS'.           'TO'.
69   'TOKEN'.      'TOKENS'.
70
71 TOKENS
72     '='. '+'. '-'. '|'. '.'. '('. ')'. '['. ']'. '{'. '}'. '..'.
73
74 TOKEN CLASSES
75   ident <|string& s|>          ALIAS 'identifier'
76     = letter { letter
77              | digit
78              | '_' }           LEX <| s = string(tokenStr); |>
```

```
79    .
80
81    str <|string& s|>                ALIAS 'string'
82      = ( "'" noQuote1 {
83              noQuote1 } "'"         /*singly quoted string: '...'*/
84        | '"' noQuote2 {
85              noQuote2 } '"'         /*doubly quoted string: "..."*/
86        )                            LEX <|
87                                      string tmp = string(tokenStr);
88                                      s = tmp.substr(1, tmp.length()
89                                                     - 2);
90                                     |>
91    .
92
93    num <|int& n|>                   ALIAS 'number'
94      = digit { digit }              LEX <| n = atoi(tokenStr); |>
95    .
96
97    src <|string& s|>                ALIAS 'source'
98      = '<|' | '^^'                  LEX <|
99                                       s = HandleSourceCode(yylloc);
100                                    |>
101   .
102
103 PRAGMAS
104   condDefine
105     = '$DEF'    /*ident<|&v|>*/    LEX <| HandleDEF(*yylloc); |>
106   .
107
108   condUndefine
109     = '$UNDEF' /*ident<|&v|>*/     LEX <| HandleUNDEF(*yylloc); |>
110   .
111
112   condIf
113     = '$IF'     /*ident<|&v|>*/    LEX <| HandleIF(*yylloc); |>
114   .
115
116   condElse
117     = '$ELSE'                      LEX <| HandleELSE(*yylloc); |>
118   .
119
120   condEnd
121     = '$END'                       LEX <| HandleEND(*yylloc); |>
122   .
123
124 NONTERMINALS
125     Cocol4BoB.        SetDecl.        ChrSet.
126     Special.          CommentDecl.    Delimiter.
127     DelimiterPart.    KeywordDecl.    TokenDecl.
```

```
128     TokenClassDecl.   PragmaDecl.     NonTerminalDecl.
129
130     AliasName<|string &alias|>.
131
132     LexExpr.          LexTerm.        LexFact.
133     RuleDef.          SynExpr.        SynTerm.
134     SynFact.
135
136     LexDecl   <|string &src|>.
137     LexAction <|string &src|>.
138     SemDecl   <|string &src|>.
139     SemAction <|string &src|>.
140     PreAction <|string &src|>.
141     PostAction<|string &src|>.
142
143 RULES
144   Cocol4BoB =
145     'COMPILER' ident<|str|>     SEM<| /* SetCompilerName */ |>
146     [ 'LEX' { src<|str|>        SEM<| /* SetUserCode */ |> } ]
147     [ 'SEMDECL' { src<|str|>    SEM<| /* SetUserDecl */ |> } ]
148     [ 'SEM' { src<|str|>        SEM<| /* SetUserCode */ |> } ]
149     [ 'IGNORE' 'CASE'     SEM<| /* SetOptionIgnoreCase */ |> ]
150     [ 'CHARACTER' 'SETS'    { SetDecl     } ]
151     [ 'COMMENTS'            { CommentDecl } ]
152     [ 'KEYWORDS'            { KeywordDecl } ]
153     [ 'TOKENS'              { TokenDecl   }
154                               SEM<| /* EndTokenDecl  */ |> ]
155     [ 'TOKEN' 'CLASSES'     { TokenClassDecl    } ]
156     [ 'PRAGMAS'             { PragmaDecl        } ]
157     [ 'NONTERMINALS'        { NonTerminalDecl   } ]
158     [ 'RULES'               { RuleDef           } ]
159     'END' ident<|str|> '.'
160   .
161
162   SetDecl =             LOCAL<| string ident = "";         |>
163     ident<|str|> '='    SEM<| ident = str;                 |>
164     ChrSet {
165     ( '+'               SEM<| /* SetDeclSetSign - PLUS  */ |>
166     | '-'               SEM<| /* SetDeclSetSign - MINUS */ |>
167     ) ChrSet }
168     [ 'IGNORE'          SEM<| /* SetDeclSetIgnore -true */ |>
169     ] '.'               SEM<| /* AddSetDecl             */ |>
170   | RECOVER BY '.'
171   .
172
173   ChrSet =
174     ident<|str|>        SEM<| /* SetDeclAddIdent       */ |>
175   | 'ANY'               SEM<| /* SetDeclAddANY         */ |>
176   | 'EOL'               SEM<| /* SetDeclAddEOL         */ |>
```

```
177   | Special [ '..'          SEM<| /* SetDeclAddRange       */ |>
178     Special ]               SEM<| /* SetDeclEndSpecial     */ |>
179   .
180
181   Special =
182     str<|str|>              SEM<| /* SetDeclAddSpecial  */ |>
183   | 'CHR' '('
184     num<|num|> ')'          SEM<| /* SetDeclAddSpecial  */ |>
185   .
186
187   CommentDecl =
188     'FROM' Delimiter        SEM<| /* CmntFromEnd       */ |>
189     'TO' Delimiter
190     [ 'NESTED'              SEM<| /* CmntSetNested     */ |>
191     ]
192         [ 'HANDLER'
193       ident<|str|>          SEM<| /* CmntSetHandler    */ |>
194     ] '.'                   SEM<| /* AddCmnt           */ |>
195   | RECOVER BY '.'
196   .
197
198   Delimiter =
199     DelimiterPart
200     [ DelimiterPart ]
201   .
202
203   DelimiterPart =
204     str<|str|>              SEM<| /* CmntDelimAddStr   */ |>
205   | ident<|str|>            SEM<| /* CmntDelimAddIdent */ |>
206   | 'CHR' '('
207     num<|num|> ')'          SEM<| /* CmntDelimAddChr   */ |>
208   | 'EOL'                   SEM<| /* CmntDelimAddEOL   */ |>
209   .
210
211   KeywordDecl =             LOCAL<|
212                               string keyword = "";
213                               string alias = "";
214                             |>
215   ( ident<|str|>            SEM<| keyword = str; |>
216     [ AliasName<|alias|> ]  SEM<| /* AddKeyword/Alias  */ |>
217       '=' str<|str|>        SEM<| /* AddKeywordAlias   */ |>
218     { '|' str<|str|>        SEM<| /* AddKeywordAlias   */ |>
219     }                       SEM<| /* AddKeyword        */ |>
220   | str<|str|>              SEM<| keyword = str;
221                                   /* AddKeywordAlias   */
222                             |>
223     [ AliasName<|alias|> ]  SEM<| /* AddKeyword/Alias  */ |>
224   ) '.'
225   | RECOVER BY '.'
```

```
226   .
227
228   TokenDecl =                 LOCAL<|
229                                 string token = "";
230                                 string alias = "";
231                               |>
232   ( ident<|str|>              SEM<| token = str; |>
233     [ AliasName<|alias|>
234     ] '=' LexExpr             SEM<| /* AddIdentToken     */ |>
235   | str<|str|>                SEM<| /* AddStringToken    */ |>
236   ) '.'
237   | RECOVER BY '.'
238   .
239
240   TokenClassDecl =            LOCAL<|
241                                 string ident = "";
242                                 string alias = "";
243                                 string param = "";
244                                 string decl = "";
245                                 string src = "";
246                               |>
247     ident<|str|>              SEM<| ident = str; |>
248     { src<|str|>              SEM<| param += str; |>
249     } [ AliasName<|alias|>
250     ] '='
251     LexExpr
252     [ LexDecl<|decl|> ]
253     [ LexAction<|src|> ]
254     '.'                       SEM<|
255                                 /* AddTokenClass      */
256                                 /* AddKeyword         */
257                                 /* AddExternalLexDecl */
258                               |>
259   | RECOVER BY '.'
260   .
261
262   PragmaDecl =                LOCAL<|
263                                 string ident = "";
264                                 string decl = "";
265                                 string src = "";
266                               |>
267     ident<|str|> '='          SEM<| ident = str; |>
268     LexExpr
269     [ LexDecl<|decl|> ]
270     [ LexAction<|src|> ]
271     '.'                       SEM<| */ AddPragma         */ |>
272   | RECOVER BY '.'
273   .
274
```

```
275  NonTerminalDecl =          LOCAL<|
276                               string ident = "";
277                               string params = "";
278                             |>
279    ident<|str|>             SEM<| ident = str;            |>
280    { src<|str|>             SEM<| params += str;          |>
281    } '.'                    SEM<| /* AddNTDecl         */ |>
282  | RECOVER BY '.'
283  .
284
285  AliasName<|string &alias|> =
286    'ALIAS'
287    ( ident<|str|>           SEM<| alias = str;            |>
288    | str<|str|>             SEM<| alias = str;            |>
289    )
290  .
291
292  LexExpr =
293    LexTerm { '|'            SEM<| /* AddSymbol - "|"   */ |>
294    LexTerm }
295  .
296
297  LexTerm =
298    LexFact { LexFact }
299    [ 'IF' 'FOLLOWED'
300      'BY' '('               SEM<| /* AddSymbol - "/"   */ |>
301    LexExpr ')' ]
302  .
303
304  LexFact =
305    ( ident<|str|>           SEM<| /* AddIdent          */ |>
306    | str<|str|>             SEM<| /* AddString         */ |>
307    | 'CHR' '('
308      num<|num|> ')'         SEM<| /* AddChar           */ |>
309    | 'EOL'                  SEM<| /* AddSymbol - "\n"  */ |>
310    | '('                    SEM<| /* AddSymbol - "("   */ |>
311      LexExpr ')'            SEM<| /* AddSymbol - ")"   */ |>
312    | '['                    SEM<| /* AddSymbol - "("   */ |>
313      LexExpr ']'            SEM<| /* AddSymbol - ")?"  */ |>
314    | '{'                    SEM<| /* AddSymbol - "("   */ |>
315      LexExpr '}'            SEM<| /* AddSymbol - ")*"  */ |>
316    )
317  .
318
319  RuleDef =                  LOCAL<|
320                               string ident = "";
321                               string param = "";
322                               string semDecl = "";
323                               string preAction = "";
```

```
                              string postAction = "";
                            |>
  ident<|str|>              SEM<| ident = str; |>
  { src<|str|>              SEM<| param += str; |>
  } '='
  [ SemDecl<|semDecl|> ]
  [ PreAction<|preAction|> ]
  [ PostAction<|postAction|> ]
                            SEM<| /* StartRule         */ |>
  SynExpr '.'               SEM<| /* EndRule           */ |>
| RECOVER BY '.'
.

SynExpr =
  SynTerm                   SEM<| /* EndProduction/
                                     EndCaseExpr       */ |>
  { '|' SynTerm             SEM<| /* EndProduction/
                                     EndCaseExpr       */ |>
  }
.

SynTerm =                   SEM<| /* StartCaseExpr/
                                     StartProduction   */ |>
  SynFact { SynFact }
.

SynFact =                   LOCAL<|
                              string ident = "";
                              string param = "";
                              string sem = "";
                              string syn = "";
                            |>
( ident<|str|>              SEM<|
                              ident = str;
                              /* AddIdent */
                            |>
  { src<|str|>              SEM<| param = str;           |>
  }                         SEM<| /* AddIdent         */ |>
| str<|str|>                SEM<| /* AddString        */ |>
| SemAction<|sem|>          SEM<| /* AddSem           */ |>
| '('                       SEM<| /* StartCompExpr    */ |>
  SynExpr ')'               SEM<| /* EndCompExpr      */ |>
| '['                       SEM<| /* StartOptExpr     */ |>
  SynExpr ']'               SEM<| /* EndOptExpr       */ |>
| '{'                       SEM<| /* StartLoopExpr    */ |>
  SynExpr '}'               SEM<| /* EndLoopExpr      */ |>
| 'EPS'                     SEM<| /* AddEPS           */ |>
| 'RECOVER' 'BY'            SEM<| /* AddRecoverBy     */ |>
| 'ANY'                     SEM<| /* not supported    */ |>
```

```
373   )
374   .
375
376   LexDecl<|string &src|>  =     SEM<| src = "";   |>
377     'LEXLOCAL' { src<|str|>     SEM<| src += str; |> }
378   .
379
380   LexAction<|string &src|> =    SEM<| src = "";   |>
381     'LEX' { src<|str|>          SEM<| src += str; |> }
382   .
383
384   SemDecl<|string &src|> =      SEM<| src = "";   |>
385     'LOCAL' { src<|str|>        SEM<| src += str; |> }
386   .
387
388   SemAction<|string &src|> =    SEM<| src = "";   |>
389     'SEM' { src<|str|>          SEM<| src += str; |> }
390   .
391
392   PreAction<|string &src|> =    SEM<| src = "";   |>
393     'PRE' { src<|str|>          SEM<| src += str; |> }
394   .
395
396   PostAction<|string &src|> =   SEM<| src = "";   |>
397     'POST' { src<|str|>         SEM<| src += str; |> }
398   .
399
400 END BoB.
```

References

[AMT90] Rieks op den Akker, Borivoj Melichar, and Jorma Tarhio. "The Hierarchy of LR-Attributed Grammars". In: *Proceedings of the International Conference WAGA on Attribute Grammars and their Applications*. London, UK, UK: Springer-Verlag, 1990.

[Aho+06] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[Bac+63] J. W. Backus et al. "Revised report on the algorithm language ALGOL 60". In: *Communications of the ACM* 6.1 (1963). Ed. by P. Naur.

[DFK05] J. D'Anjou, S. Fairbrother, and D. Kehn. *The Java Developer's Guide to Eclipse*. 2. Edition. Addison-Wesley, 2005.

[DP90] H. Dobler and K. Pirklbauer. "Coco-2: A New Compiler Compiler". In: *SIGPLAN Notices* 25.5 (1990).

[DP91] H. Dobler and K. Pirklbauer. "Top-Down Parsing in Coco-2". In: *SIGPLAN Notices* 26.3 (1991).

[DS11] C. Donnelly and R. Stallman. *Bison*. 2.5. Free Software Foundation, Inc. 2011.

[Dic11] W. Dichler. "Teil 1: Syntax Highlighting für einen Pseudocode-Editor in Eclipse". Bachelor Thesis. Hagenberg, Austria: Fachhochschule O.Ö., Studiengang Software Engineering, 2011.

[Dob10] H. Dobler. *Compiler- und Werkzeugbau*. Technical Report. FH O.Ö., Studiengang Software Engineering, 2010.

[Dob11] H. Dobler. *Formale Sprachen und Automatentheorie*. Technical Report. FH O.Ö., Studiengang Software Engineering, 2011.

[ISO96] ISO. *Information Technology – Syntactic Metalanguage – Extended BNF*. Norm ISO/IEC 14977:1996(E). International Organization for Standardization, 1996.

[Joh75] Stephen C. Johnson. *Yacc: Yet Another Compiler-Compiler*. Technical Report. Bell Laboratories, 1975.

[KRT72] B. W. Kernighan, D. M. Ritchie, and K. L. Thompson. *QED text editor*. Computing Science Technical Report. Bell Laboratories, 1972.

[Kat83] J. van Katwijk. "A preprocessor for YACC or a poor man's approach to parsing attributed grammars". In: *SIGPLAN Not.* 18.10 (1983).

[Kle56] S. C. Kleene. "Representation of Events in Nerve Nets and Finite Automata". In: *Automata Studies* (1956).

[LS75] M. E. Lesk and E. Schmidt. *Lex – A Lexical Analyzer Generator*. Technical Report. Bell Laboratories, 1975.

[Lev09] J. Levine. *flex & bison*. First Edition. O'Reilly Media, Inc., 2009.

[MH10] K. Martin and B. Hoffman. *Mastering CMake*. Kitware Incorporated, 2010.

[Mös87] H. Mössenböck. *Compilererzeugende Systeme für Mikrocomputer*. Dissertation der Johannes-Kepler-Universität Linz. Wien: Verlag VWGÖ, 1987.

[PEM07] V. Paxson, W. Estes, and J. Millaway. *Flex*. 2.5.35. The Flex Project. 2007.

[Pra11] S. Prata. *C++ Primer Plus*. Developer's Library. Pearson Education, 2011.

[SSS90] S. Sippu and E. Soisalon-Soininen. *Parsing Theory – Volume II: LR(k) and LL(k) Parsing*. EATCS Monographs on Theoretical Computer Sciences: European Association for Theoretical Computer Science. Springer, 1990.

[Wir77] Niklaus Wirth. "What can we do about the unnecessary diversity of notation for syntactic definitions?" In: *Communications of the ACM* 20.11 (1977).

[Wir85] N. Wirth. *Programming in Modula-2*. 3rd Edition. Texts and Monographs in Computer Science. Springer-Verlag, 1985.

[Zer97] D. Umrigar Zerksis. *Zyacc – A Parser-Generator*. 1.0. 1997.

[TIO13] TIOBE Software. *TIOBE Programming Community Index*. 2013. URL: http://www.tiobe.com/tpci.htm.